UNIVERSITETET I OSLO
Institutt for informatikk

Evaluering av CORBA-implementasjoner på håndholdte enheter i trådløse nettverk

Lars Preben S. Arnesen
Eirik Valen

November 2002

Forord

Denne rapporten er resultatet av vårt hovedfagsstudium ved Institutt for informatikk, Universitetet i Oslo, i perioden 2000-2002.

Først og fremst vil vi rette en stor takk til våre veiledere, Tom Kristensen og Thomas Plagemann, som har vært meget behjelpelige gjennom hele perioden.

Vi vil også takke Anders Breivik, Benedikte Lund, Martin Thornquist, Per Kristian Gjermshus, samt våre foreldre, for moralsk støtte og hjelp med korrekturlesing.

Oslo, 1. november 2002

Eirik Valen & Lars Preben Arnesen

Innhold

Tabeller

Figurer

Programeksempler

Kapittel 1

Innledning

Det er ikke mangel på visjoner om morgendagens digitale tjenester. Lenge har *science fiction*-genren beskrevet fullautomatiserte hus, styrt av intelligente systemer. Hvem har vel ikke forestilt seg hvor praktisk det hadde vært å kunne skru av komfyren fra kontoret? Eller hva med en digital assistent, som tilbyr deg full kontroll over hjemmekinoanlegget i det du går inn i stuen? Med enheter tilknyttet et felles nettverk, åpner det seg en rekke muligheter for nye, digitale tjenester.

I de siste årene har vi vært vitne til en enorm utvikling innenfor trådløse, mobile enheter. Bruken av GSM-telefoner har eksplodert siden midten av 90-tallet, og har blitt en viktig del av infrastrukturen for kommunikasjon i Europa. Mobiltelefonteknologien har gitt oss en ny måte å kommunisere på, og det dukker stadig opp nye tjenester — tjenester som går langt utover det som var påtenkt da kommunikasjonsprotokollene ble utviklet. Denne revolusjonen har medført kraftig videreutvikling av de trådløse nettverkene, og vi er nå ved overgangen til tredje generasjons mobilnett. Utviklingen har gått fra analoge til digitale signaler, fra enkel meldingstjeneste til nettsurfing, og fra enkel tale til multimedia.

I tillegg til utviklingen av mobiltelefonteknologien ser vi også en utvikling innen datakommunikasjon mellom andre typer elektroniske enheter. Fra kommunikasjon via infrarøde signaler, og kabler med proprietære kontakter, ser det nå ut til at trenden beveger seg mot trådløs kommunikasjon over kortdistanse-radionett. Bluetooth er standarden de største elektronikkprodusentene nå kaster seg over; stadig flere mobiltelefonmodeller blir utstyrt med Bluetooth, og nye produkter dukker opp fortløpende. I skrivende stund finnes det tastaturer, håndfrisett for mobiltelefoner, prin-

terløsninger, og ikke minst håndholdte datamaskiner, som støtter denne nye kommunikasjonsteknologien.

Mot slutten av 90-tallet økte salget av håndholdte datamaskiner, og det finnes i dag en lang rekke leverandører og modeller. Til nå har disse vært utstyrt med kommunikasjonsporter beregnet på datasynkronisering mot arbeidsstasjon, mobiltelefon eller andre håndholdte enheter. Vi ser nå en overgang mot internettbaserte tjenester ved at de håndholdte enhetene utstyres med både program- og maskinvare som gjør dem i stand til å knytte seg opp mot Internett.

Dagens håndholdte modeller, og mest sannsynlig de som slippes de nærmeste årene, har langt mindre ressurser enn stasjonære maskiner, både hva gjelder prosessor og minne, og ofte også nettverk. Det vil derfor være hensiktsmessig, og i noen tilfeller helt nødvendig, å tilpasse tjenestene til de håndholdte enhetene. En løsning er å benytte seg av *distribuert databehandling* i form av en klient/tjener-modell, hvor den håndholdte enheten sender en forespørsel til en kraftigere tjener, som utfører en operasjon og returnerer resultatet. Dette er ingen ny tankegang, og har blitt implementert på en rekke måter de siste tiårene.

Applikasjonsutviklingen har i det siste tiåret vært dominert av de programmeringsspråk som støtter *objektorientering*. Dette konseptet ble først introdusert med Simula 67 i 1967, og i dag har vi flere språk som støtter dette paradigmet, blant annet C++ og Java. *Mellomvare* er et begrep som brukes om et lag som ligger mellom applikasjoner og nettverk, og gjør nettverkskommunikasjonen transparent for applikasjonslaget. Noen av de mest populære mellomvareløsningene vi har i dag baserer seg på en modell med distribuerte objekter.

Common Object Request Broker Architecture (CORBA) er en mellomvarearkitektur som gir programvareutviklere mulighet til å benytte seg av distribuerte objekter. Heterogene, distribuerte systemer består av forskjellig maskin- og programvare, og utgjør en ekstra utfordring for mellomvaren. Med CORBA kan applikasjoner på en transparent måte få tilgang til objekter på andre maskiner som om de skulle vært lokale, uavhengig av programmeringsspråk og hvilken arkitektur objektene er implementert for.

Kombinasjonen av håndholdte datamaskiner, CORBA-basert programvare og trådløse nettverk gjør det mulig å implementere områdebaserte tjenester som kan benyttes av forskjellige klienter. Enten det gjelder å velge mellom sin digitale assistent og den vanlige fjernkontrollen for å skru på

søndagsfilmen, eller å følge med på aksjekursene på en togreise, er CORBA en arkitektur som gjør slike distribuerte tjenester mulig.

1.1 Fokus og mål

Målet for denne hovedfagsoppgaven er å kartlegge om CORBA-implementasjoner er egnet til å benyttes på de håndholdte maskinene som finnes på dagens marked. Det finnes en rekke forskjellige modeller med forskjellige spesifikasjoner, både når det gjelder ytelse og nettverksgrensesnitt. Vi ønsker å undersøke nærmere hvilke modeller som egner seg til bruk av distribuerte tjenester, samt hvilke begrensninger de ulike nettverksteknologiene setter.

Denne hovedfagsoppgaven inngår i MULTE-prosjektet (Multimedia Middleware for Low Latency High Throughput Environment), som har som mål å utvikle ny form for ORB-basert multimedia-mellomvare. Håndholdte maskiner er i dag ikke utstyrt med høyhastighets nettverk, men det er i MULTE-sammenheng interessant å kartlegge hvilken ytelse man kan oppnå med dagens håndholdte løsninger. UNINETT har, igjennom UMNS-prosjektet (UNINETT Mobile Networking System), bidratt med ressurser i form av utlån av maskinvare (håndholdte enheter og Wi-Fi-utstyr) til vårt prosjekt.

I utgangspunktet hadde vi planer om å studere UIC-CORBA, en CORBA-modul for *Universally Interoperable Core*, som er utviklet for ressurssvake enheter. Etter å ha studert implementasjonen grundig, valgte vi å forkaste den på grunn av store avvik fra CORBA-standarden. I stedet valgte vi to andre CORBA-implementasjoner; én kommersiell, *Orbix/E*, og én basert på *Open Source*, *MICO*. MICO er en fullverdig CORBA-implementasjon, mens Orbix/E er optimalisert for ressurssvake enheter og implementerer kun et subsett av CORBA-spesifikasjonen.

CORBA medfører en del *overhead* i forhold til bruk av regulære objekter. Kartlegging av ressurs- og tidsforbruket ved bruk av CORBA-objekter er derfor meget interessant for design av distribuerte tjenester — spesielt dersom applikasjonen krever lav responstid for operasjonene de distribuerte objektene skal utføre. Responstiden til et CORBA-objekt er avhengig av flere faktorer, og må også sees i lys av hvilken type maskinvare og nettverksteknologi som benyttes. Enkle metodekall vil hovedsakelig bli påvirket av nettverkslatensen og CORBA-implementasjonen, mens responsti-

den for overføring av større datamengder i tillegg vil påvirkes av nettverkets gjennomstrømning og CORBA-objektenes ytelse ved datatransformasjon.

En CORBA-implementasjon må dessuten kunne håndtere mange objekter uten at responstiden blir uakseptabelt høy, og det er derfor interessant å ta i betraktning skaleringsevnen for antall objekter. Naturlig nok påvirkes også responstidene av maskinvaren, og vi vil derfor benytte oss av flere forskjellige maskinvarekonfigurasjoner i tillegg til ulike nettverk og CORBA-implementasjoner.

Utviklingen for maskin- og programvare beveger seg i en enorm fart og produkter og teknikker blir kontinuerlig forbedret. Til tross for at produktene vi tar for oss i denne oppgaven mest sannsynlig ikke finnes på markedet om få år, vil morgendagens produkter være basert på erfaringer gjort fra dagens versjoner, og studier som våre vil derfor forhåpentligvis være av nytte.

1.2 Arbeidsmetoder

En stor del av denne hovedoppgaven har bestått i å studere og tilpasse ulike programvareløsninger, enten det har dreid seg om CORBA-implementasjoner, analyseverktøy eller operativsystemer. Det meste av programvaren vi har benyttet er *Open Source*, og i kontinuerlig utvikling. I motsetning til kommersielle produkter, som gjerne lanserer nye versjoner med flere måneders mellomrom, kan man for *Open Source*-programvare ofte få tilgang til uferdige utviklingsversjoner. Et vanlig fenomen knyttet til dette er at *Open Source*-prosjektene gjerne sliter med å holde dokumentasjonen oppdatert. Orbix/E og MICO er et godt eksempel på nettopp dette: Orbix/E leveres med fyldig, oppdatert dokumentasjon, mens MICO er relativt sparsommelig dokumentert. For sistnevnte er informasjonen i tillegg noe utdatert.

Open Source-miljøet er imidlertid kjent for å yte gratis hjelp til alle som viser interesse for prosjektene. Under arbeidet med oppgaven har vi brukt en rekke e-postlister og nyhetsgrupper som supplement til den dokumentasjonen vi har hatt tilgjengelig. For *Open CORBA Benchmarking* og *UIC-CORBA* er dokumentasjonen meget tynn, og ingen av prosjektene har hatt en brukermasse av slik størrelse at den har utgjort et miljø vi har kunnet henvende oss til. Vi har imidlertid fått god hjelp fra utviklerne.

I tillegg til å bruke nyhetsgrupper og e-postlister har vi benyttet bøker, vitenskapelige artikler og en rekke websider. Artikkelreferanser, samt pekere til sentrale websider, er gjengitt i tillegget.

De fleste ord og uttrykk innen dataverdenen er engelske og all litteratur vi har benyttet oss av er også engelskspråklig. I denne oppgaven har vi benyttet norske ord og uttrykk så langt som mulig, men vi har ikke klart å finne passende oversettelser i alle sammenhenger, og har da i stedet valgt å bruke de engelske uttrykkene. Vi har også valgt å ikke oversette sitater. Kildekoden til programmene vi har skrevet, inneholder utelukkende engelske variabel- og metodenavn for å gjøre den forståelig for andre, engelskspråklige interessenter.

For å nå målet har vi gjennomført ytelsesmålinger for forskjellige kombinasjoner av program- og maskinvare. Vi satte først opp forskjellige scenarier for de forskjellige kombinasjonene av maskinvare, som vi så prøvde å benytte til ytelsesmålinger for begge CORBA-implementasjonene. Ikke alle scenarier ble benyttet, hovedsaklig fordi vi støtte på problemer med program- og maskinvare, som ville tatt lengre tid å løse enn vi hadde til rådighet.

Etterhvert som vi utførte ytelsesmålingene, dukket nye problemstillinger og spørsmål opp. Dette medførte at vi gjorde flere målinger — enten dupliserte målinger for å bekrefte eller avkrefte uventede resultater, eller nye kombinasjoner for å belyse aspekter vi ikke dekket med de planlagte testkombinasjonene.

Vi har delt analysen opp i tre hoveddeler: CORBA-implementasjoner, maskinvare og nettverksteknologier. Vi har fokusert på gjennomsnittsmålingene *Open CORBA Benchmarking* gir, og benyttet data fra egne systemforbruk-målinger for å belyse eventuelle avvik. Hovedsaklig har vi studert grafiske fremstillinger av måleresultatene, men ved detaljstudier av enkelte målinger har vi benyttet oss av enkeltverdier. Grunnet *Open CORBA Benchmarking*s behandling av rådataene i form av beregning av gjennomsnitts-, minimum- og maksimumsverdier har vi ikke hatt mulighet til å beregne standardavvik for å skille ut eventuelle utliggere. Vi valgte derfor å fokusere på gjennomsnittsverdiene, og benyttet kun minimum- og maksimumsverdier til å belyse avvik i gjennomsnittsverdiene.

1.3 Oppgavens oppbygging

Oppgaven er delt inn som følger: I kapittel 2 gir vi en beskrivelse av de forskjellige teknologiene som benyttes i oppgaven, mens kapittel 3 omhandler ytelsesanalyser av CORBA-implementasjoner. Kapittel 4 inneholder konkrete mål for analysene, kravene til maskinvare og programvare, samt beskrivelse av testkonfigurasjonene. Program- og maskinvaren vi har valgt å bruke til analysene er beskrevet nærmere i kapittel 5. Egenproduserte programmer og tilpasninger av program- og maskinvare brukt under målinger og analysen er beskrevet i kapittel 6.

Kapittel 7, analysen av de innsamlede dataene, er det største kapittelet i oppgaven, og består av tre hoveddeler med forskjellig fokus: CORBA-implementasjoner, maskinvare og nettverk. I første delkapittel ser vi på CORBA-implementasjoner, mens vi i maskinvare-delen sammenlikner de forskjellige maskinene og forskjellige klient-/tjener-konfigurasjoner. Tredje delkapittel omhandler de trådløse nettverkenes innflytelse på ytelsen til CORBA-implementasjonene. Avslutningsvis ser vi på ressursforbruket under målingene, resultater fra beslektede forsøk, og gir et sammendrag av resultatene. I kapittel 9 avrundes oppgaven med en evaluering av målene, resultater og forslag til videre arbeid.

Kapittel 2

Bakgrunn

For å kunne gjennomføre ytelsesanalyser av CORBA-implementasjoner på håndholdte enheter, er vi avhengig av forskjellig maskin- og programvare. Dette kapittelet tar for seg de teknologiene og produktgruppene vi har brukt i oppgaven.

2.1 Utvikling innen håndholdte datamaskiner

Det finnes et bredt spekter av håndholdte enheter på dagens marked. Felles for alle er at de selges med programvare som skal erstatte en vanlig kalender-planlegger; kalender, notatblokk og telefonliste er den vanligste funksjonaliteten. I tillegg leveres noen med regneark og tekstbehandler. Disse håndholdte enhetene er også tilrettelagt for å synkronisere dataene med tilsvarende programvare for arbeidsstasjoner, slik at man ikke trenger å oppdatere flere kalendere, telefonlister og liknende. De håndholdte enhetene kan deles inn i to hovedkategorier: ressurssterke og ressurssvake enheter.

2.1.1 Ressurssvake enheter

US Robotics[1] slapp sin første håndholdte modell, "Palm Pilot 1000", i 1996, og siden har nye Palm-modeller kommet fortløpende ut på markedet. Fi-

[1]Palm-modellene ble først produsert av US Robotics, som senere ble kjøpt av 3Com. I 2000 ble Palm Inc. og US Robotics skilt ut fra 3Com igjen.

losofien til Palm Inc. har vært å lage håndholdte enheter som gjør akkurat det brukerne trenger og ingenting mer. Batterilevetid har vært prioritert fremfor prosessorkraft og høy minnekapasitet. Ved å benytte en ressurssvak prosessor, minimalt med minne og gråtone-LCD-skjerm[2], har Palm Inc. klart å levere en rekke enheter som holder flere uker uten oppladning ved *normal* bruk. Denne filosofien medfører imidlertid at programmene som kjøres på enhetene bør være så små og effektive som mulig. I første kapittel av boken "Palm Programming: The Developer's Guide"[2] står det:

> «The Palm is a lively little machine, so don't bog it down with slow apps. [...] It has a pip-squeak processor with no more power than a desktop machine in the mid-1980s. As a result, you should precalculate as much as possible on the desktop.»

De siste årene har også flere maskinvareprodusenter kastet seg inn på markedet for håndholdte maskiner. Noen av dem har valgt å lisensiere operativsystemet til Palm Inc. (PalmOS), som frem til versjon 4 kun har kjørt på Motorola 68000-prosessorer. Denne prosessorserien har relativt lav ytelse, og det har gjort at alle enhetene som har benyttet seg av PalmOS har havnet i samme ytelseskategori.

2.1.2 Ressurssterke enheter

I kategorien for de kraftige håndholdte enhetene, er Intel og Microsoft store aktører. Intels StrongARM SA-1110[3] brukes i flesteparten av de håndholdte enhetene, og operativsystemet som benyttes er hovedsakelig Windows CE. StrongARM SA-1110-brikken leveres i utgaver med klokkefrekvens fra 100 MHz til 233 MHz. Basert på CNets maskinvareoversikt synes det som om majoriteten av dagens kraftige håndholdte enheter bruker 206 MHz-modellen av SA-1110.

I motsetning til de ressurssvake enhetene, har SA-1110 kapasitet til mer CPU-intensive oppgaver, som for eksempel dekomprimering av audio og video. Mediefiler er generelt sett store, og dette krever større lagringsplass enn det som er normalt på de ressurssvake enhetene. De ressurssterke enhetene er som regel utstyrt med 32-64MB RAM.

[2]Enkelte av de nyere maskinene leveres også med fargeskjerm.

[3]Opprinnelig utviklet av Digital Equipment Corporation, senere overtatt av Intel.

Fargeskjermer er også mest utbredt på de kraftige enhetene, og bakgrunnsbelysning på slike LCD-skjermer krever mye strøm. Rask prosessor og mye minne bidrar også til å øke strømforbruket, som er relativt høyt i forhold til forbruket på de ressurssvake enhetene.

2.1.3 Videre utvikling

Intel har lansert arvtakeren til StrongARM-serien. Den nye prosessorserien heter *Xscale*, og er allerede tatt i bruk i den nye iPAQ-serien. iPAQ 3950/3970 bruker PXA250 — en 400 MHz Xscale-prosessor. Neste versjon av PalmOS (5) støtter Intels Xscale/ARM-arkitektur og det er derfor naturlig å anta at det kommer til å bli sluppet flere PalmOS-baserte håndholdte enheter som er i samme ressursklasse som dagens Windows CE-modeller.

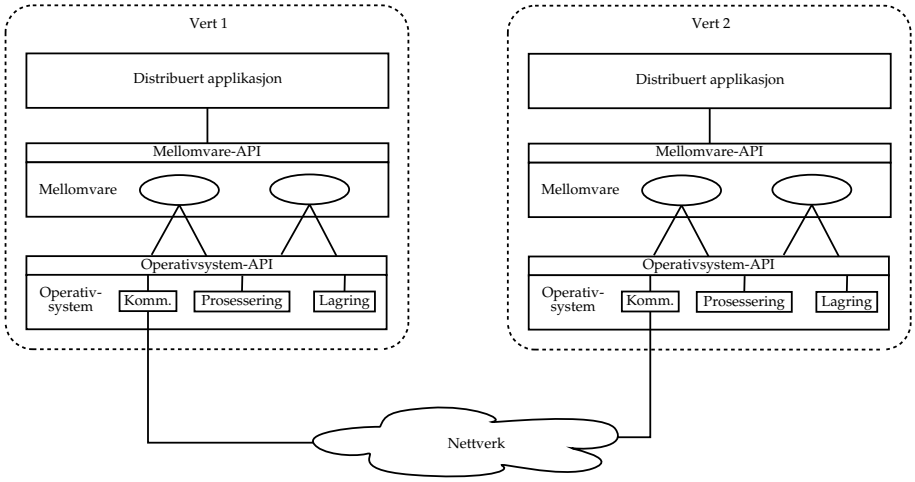
Det ser ut til at utviklingen stadig går i retning av kraftigere modeller, ettersom prosessorene basert på Xscale-arkitekturen er raskere enn de basert på StrongARM-arkitekturen. Microsoft og Intel har inngått samarbeid om videre bruk av Xscale-arkitekturen til *Windows Media*, og det er ingenting som tyder på at behovet for prosessorkraft blir noe mindre fremover. Dersom konsumentene velger de ressurssterke enhetene fremover, er det naturlig å forvente at kategorien for ressurssvake håndholdte maskiner forsvinner.

2.2 Mellomvare

Bruk av klassebiblioteker og/eller andre former for gjenbruk av kode er en selvfølge for de aller fleste programmerere. Uten slike hjelpemidler ville selv relativt enkle programmeringsoppgaver bli særdeles tidkrevende. Et distribuert system bygger blant annet på en rekke nødvendige mekanismer for kommunikasjon mellom de forskjellige elementene, som ikke nødvendigvis er trivielle å implementere.

Mellomvarens viktigste rolle er å håndtere kompleksiteten og heterogeniteten i et distribuert system. På denne måten forenkles utviklingsmiljøet applikasjonsprogrammereren benytter seg av[3]. Figur 2.1 viser hvor mellomvaren ligger i forhold til applikasjoner og operativsystem.

Det finnes flere forskjellige kategorier av mellomvare. I [4] beskriver Bakken følgende kategorier: *distribuerte tupler* (Linda, Jini, SQL), *remote pro-*



Figur 2.1: Illustrasjon av mellomvarelagets plassering[3].

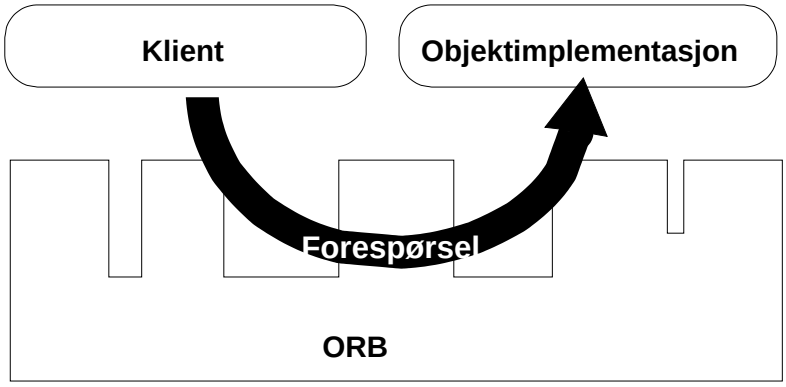
cedure call, *meldingsorientert mellomvare* og *distribuerte objekter* (CORBA, DCOM, SOAP). I denne oppgaven har vi fokusert på distribuerte objekter i form av CORBA.

2.3 CORBA

Første versjon av Common Object Request Broker Architecture (CORBA) ble lansert i 1991 av Object Management Group (OMG). Denne har blitt videreutviklet gjennom hele nittitallet, og siste versjon (3.0) ble sluppet i september 2002.

OMG ble i 1989 stiftet av 3Com, American Airlines, Canon, Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems og Unisys for å adressere problemene med å utvikle portable, distribuerte applikasjoner for heterogene systemer. OMG har i dag over 800 medlemmer, og er verdens største programvare-samarbeidsorgan[5][6]. I tillegg til CORBA har OMG også utviklet en rekke andre spesifikasjoner, deriblant UML.

CORBA-arkitekturen gjør det mulig å dele objekter i et distribuert, heterogent miljø. Ved hjelp av *Interface Definition Language* (IDL) kan man beskrive objektgrensesnitt, som videre kan brukes til generering av arkitekturspesifikke objektgrensesnitt. På denne måten gjør CORBA det mulig å benytte seg av objekter uavhengig av arkitektur.



Figur 2.2: ORB, klient og objektimplementasjon[1].

2.3.1 Oppbygning

CORBA-arkitekturen består av flere elementer. Figur 2.2 viser i grove trekk hvordan kommunikasjonen mellom klient og objektimplementasjon foregår. De neste delkapitlene beskriver de forskjellige delene av arkitekturen i detalj.

Object Request Broker

Object Request Broker (ORB) kan oversettes til *objektmegler*, og er den sentrale delen i et CORBA-system. Dokpros norskordbok definerer ordet "megler" som: *«mellommann ved kjøp og salg, inngåelse av kontrakter o.l.»*. En ORB fungerer nettopp som en *mellommann* mellom klient og distribuerte objekter, og tar hånd om kommunikasjonen mellom partene.

I følge CORBA-spesifikasjonen[1] er ORB-en ansvarlig for alle mekanismer som kreves for å finne og forberede objektet, i tillegg til mekanismene for transport av data i forbindelse med forespørselen. ORB-en har flere grensesnitt (*interfaces*) mot omverdenen. Klienten kan gjøre direkte kall mot ORB-metoder ved hjelp av *Dynamic Invocation Interface* (DII). På samme måte kan ORB-en bruke *Dynamic Skeleton Interface* for å aksessere objektimplementasjonene. Alternativet er å benytte statiske IDL-grensesnitt, som er definert ved kompilering av applikasjonen.

Interface Definition Language

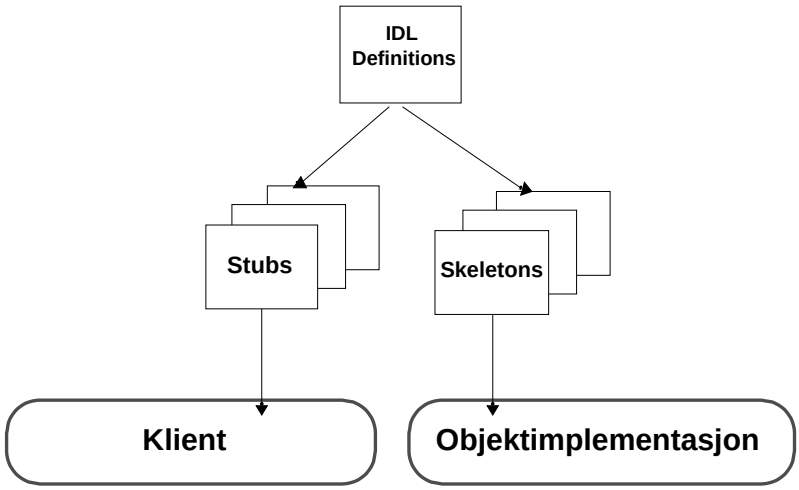
Interface Definition Language (IDL) er en ISO-standard[7] basert på OMGs arbeid. Standarden spesifiserer et definisjonsspråk for å beskrive arkitektur-uavhengige objektgrensesnitt. OMG har spesifisert oversettelser (*mapping*) for C, C++, Java, Smalltalk, COBOL, Ada, Lisp, PL/1, Python, og IDLscript. I tillegg eksisterer det også tredjeparts oversettelsespesifikasjoner.

For å generere kode fra IDL-definisjoner, benytter man en *IDL-kompilator*. IDL-kompilatoren er en implementasjon av den ovennevnte *mappingen* for det gitte språket, og genererer *stubs* og *skeletons* som grensesnitt for henholdsvis klienten og ORB-en. Objektet som skal aksesseres fra en klient implementerer *skeleton*-grensesnittet, mens *stubs*-objektene brukes som tomme grensesnitt-objekter på klientsiden; klienten ser ikke forskjell på vanlige objekter og *stubs*-objektene. Et normalt objekt eksisterer på samme maskin som klienten, mens et *stub*-objekt representerer grensesnittet til et distribuert objekt som kan befinne seg på en helt annen maskin.

IDL-kompilatoren følger som regel med CORBA-implementasjonen, og genererer implementasjonsspesifikk kode. Grensesnittet til objektene er derimot identisk for alle ORB-er, noe som gjør det mulig å aksessere objekter uavhengig av ORB og programmeringsspråk. Figur 2.3 skisserer sammenhengen mellom IDL-definisjoner, klienten og objektimplementasjonene. I tillegg til å være grensesnitt mot objektimplementasjoner, står *stubs*-objektene for *marshalling* av data. *Marshalling*-operasjonen består i å serialisere datastrukturer slik at de egner seg for overføring. De serialiserte dataene sendes til mottakeren over *Internet Inter-ORB Protocol* (beskrevet i neste avsnitt), og datastrukturene gjenoppbygges av *skeletons*-objektenes *demarshalling*-rutine. Figur 2.7 skisserer de forskjellige lagene en CORBA-forespørsel passerer igjennom.

Internet Inter-ORB Protocol

General Inter-ORB Protocol (GIOP) spesifiserer mesteparten av protokolldetaljene som er nødvendig for kommunikasjon mellom tjener og klient[6]. *Internet Inter-ORB Protocol* (IIOP) er en implementasjon av GIOP, spesifikk for TCP/IP, og spesifiserer i tillegg hvordan objektreferansene (*Interoperable Object References*, IOR) skal kodes. Både GIOP og IIOP er definert i CORBA-spesifikasjonen[1].



Figur 2.3: Generering av *stubs* og *skeletons*[1]

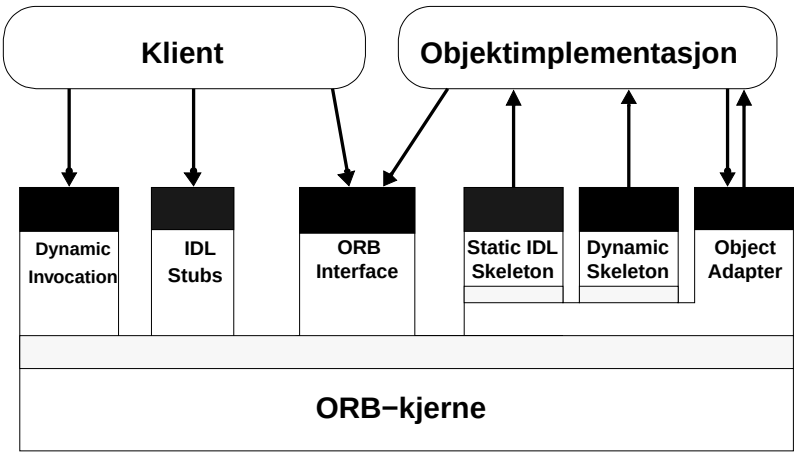
Navnetjeneste

CORBAs navnetjeneste fungerer for CORBA-objekter på samme måte som *Domain Name System* (DNS) fungerer for maskiner tilknyttet Internett[6]. Navnetjenesten oversetter objektnavn til objektreferanser (IOR), som ORB-en kan bruke for å aksessere det ønskede objektet. Tjenesten kjøres som regel separat fra ORB-en, og kan dermed sentraliseres for å benyttes av flere ORB-er.

Portable Object Adaptor

En *Object Adaptor* (OA) tar seg av opprettelse og tolkning av objektreferanser (IOR) og håndtering av objektimplementasjonene, og blir derfor grensesnittet mellom ORB-kjernen og objektimplementasjonen[1]. Prosessen med å finne frem riktig objektimplementasjon og oversende forespørslene, kalles *dispatching*.

Portable Object Adaptor (POA) er en portabel *object adaptor*, som gjør det mulig for utviklere å lage CORBA-baserte tjenerapplikasjoner som er portable mellom forskjellige CORBA-implementasjoner[8]. Den er definert i IDL, og er derfor ikke bundet opp mot ett bestemt programmeringsspråk. POA-en som grensesnitt mellom objekt-implementasjonene og ORB-en illustreres av figur 2.4.



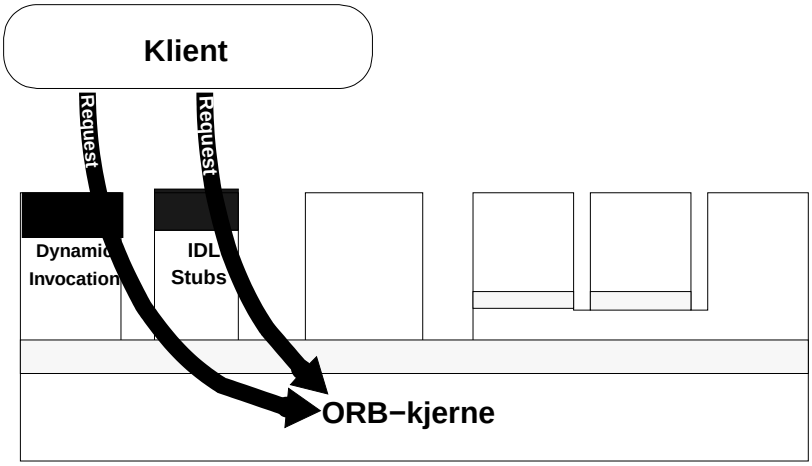
Figur 2.4: Elementene i et CORBA-system[1]

2.3.2 Forespørsel og respons med CORBA

Grensesnittene for objektimplementasjonene, *stubs* og *skeletons*, genereres utifra IDL-definisjoner og kompileres. Dersom grensesnittet ikke er kjent før kompilering, er det mulig å bruke dynamiske grensesnitt direkte mot ORB-en (DII) og/eller objektimplementasjonene (DSI). Vi beskriver videre bruk av *stubs* og *skeletons*.

Klienten oppretter et *stub*-objekt, og det foretas da et oppslag mot navnetjenesten, som returnerer en objektreferanse (IOR). Ved metodekall foretar *stub*-objektet serialisering (marshalling) av dataene før de oversendes til ORB-en (figur 2.5). Forespørslen sendes deretter til mottakeren via IIOP, over TCP/IP. Hos mottakeren formidles forespørselen til *OA*-en som finner frem korresponderede *skeleton* og objektimplementasjon. *Skeleton* utfører *demarshalling* og sender kallet, med eventuelle parameterdata, til objektimplementasjonen (figur 2.6).

Objektimplementasjonen utfører operasjonen klienten har bedt om, og sender svaret samme vei tilbake: *Skeleton* foretar *marshalling* og ORB-en sender responsen tilbake klient ORB-en. Herifra overføres responsen til *stubs*-objektet som foretar *demarshalling* og til slutt får klienten svar på me-



Figur 2.5: Kall fra klient mot ORB-en[1]

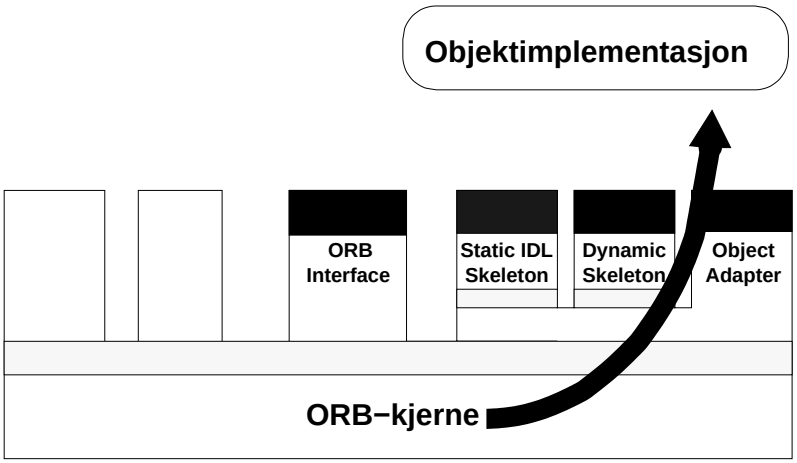
todekallet. Figur 2.7 illustrerer hvilke lag en forespørsel passerer, fra applikasjonslaget ned til nettverkslaget, og opp igjen.

2.4 Ytelsesmålinger av CORBA-implementasjoner

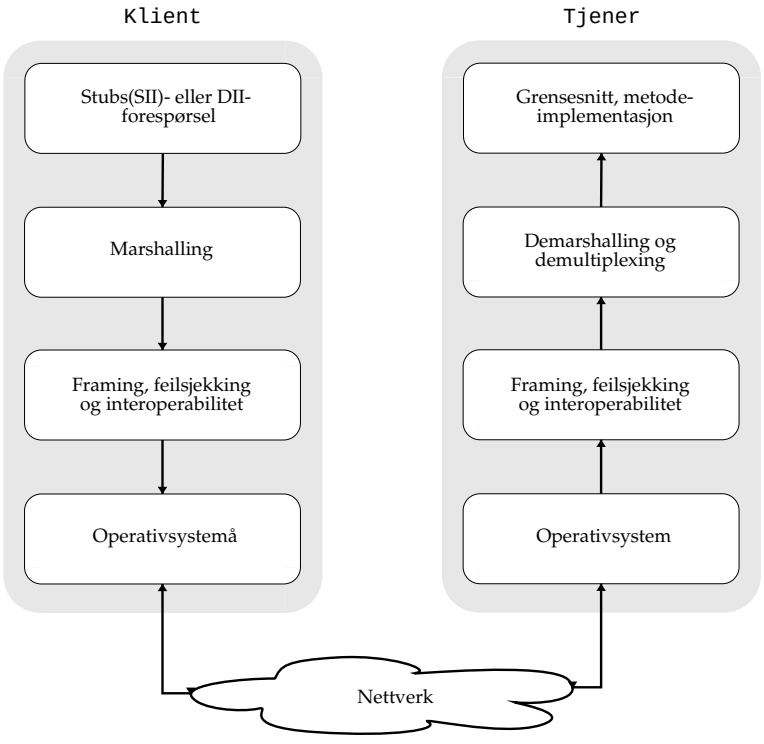
Ved ytelsesanalyser av CORBA-implementasjoner er det hovedsaklig på tre områder man foretar målinger[10]: *metodekall*, *gjennomstrømning* og *skalering*. Disse dataene sier imidlertid lite om resten av systemet ytelsesmålingene kjøres på, og det er derfor interessant å se hvor mye ressurser bruken av mellomvaren legger beslag på.

2.4.1 Metodekall

Alle metodekall blir påvirket av CORBA-laget, så den enkleste måten å måle CORBA-implementasjonens ytelse, er å måle responstiden for tomme metodekall. Med "tomme" menes at ingen parameter- eller returdata sendes og at den kalte metoden heller ikke utfører noen operasjoner. Denne testen kan sammenliknes med å måle responstiden til en maskin ved



Figur 2.6: Kall fra ORB-en mot objektimplementasjonen[1]



Figur 2.7: Illustrasjon av lagene i en CORBA-forespørsel[9].

hjelp av programmet `ping`. Andre målinger, med andre typer kall, vil med andre ord aldri få lavere responstid enn responstiden for tomme metodekall. Målinger av denne typen påvirkes i stor grad av ytelsen til nettverket mellom klient og tjener[11].

2.4.2 Gjennomstrømning

For transaksjonssystemer defineres *gjennomstrømning* som antall transaksjoner per sekund. I nettverkssammenheng defineres det som antall bits per sekund (bps)[12]. I OMGs *White Paper on Benchmarking*[10] beskrives *gjennomstrømning* som mengden arbeid som blir utført i et gitt tidsrom.

En måte å måle gjennomstrømning på er å overføre større datamengder mellom CORBA-objekter. CORBA-objektene må da kode og dekode data for overføring (IIOP), og ved å måle responstiden for slike operasjoner, er det mulig å kartlegge hvor effektiv kode IDL-kompilatoren genererer. For at målingene av gjennomstrømning skal ha relevans, må operasjonen(e) som utføres være dokumentert, slik at responstiden kan knyttes til et kjent sett referanseoperasjoner.

2.4.3 Skalerbarhet

OMGs White Paper on Benchmarking[10] omtaler to former for skalerbarhet: skalerbarhet innenfor et endesystem og distribuert skalerbarhet. Skalerbarhet innenfor et endesystem viser hvor effektivt en ORB håndterer et økende antall objekter og klientforespørsler. Distribuert skalerbarhet viser hvor mange parallelle klienter ORB-en klarer å håndtere uten at responstiden blir uakseptabelt høy.

Det er naturlig å benytte en ORB i distribuerte systemer med flere klienter og eventuelt andre ORB-er. For slike systemer av en viss størrelse, hvor man har en stor mengde objekter spredd utover flere endesystemer, vil distribuert skalerbarhet være en viktig ytelsesfaktor. Andre ytelsesaspekter ved CORBA-implementasjon må sees i lys av skalerbarheten[10].

Den enkleste måten å simulere distribuert bruk av en CORBA-applikasjon er å kjøre mange parallelle klienter som sender forespørsler til ORB-en samtidig[11]. Skalerbarheten vises i form av responstid i forhold til hvor mange klienter som sender forespørsler. En ORB som skalerer bra vil kun-

ne håndtere store mengder med parallelle forespørsler uten at responstiden blir for høy.

2.4.4 Ressursforbruk

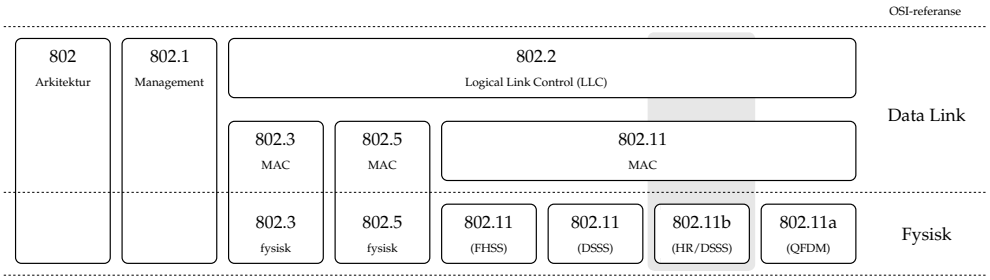
CORBA er et lag mellom applikasjons- og nettverkslaget, og dette medfører mer *overhead* enn om applikasjonen hadde kommunisert direkte med nettverkslaget. For ressursknappe miljøer, hvor prosessor-, minne- og nettverksressursene er små, er det viktig at CORBA-implementasjonen legger beslag på så lite ressurser som mulig. Ved ytelsesmålinger av CORBA-implementasjoner er det derfor interessant å benytte eksisterende målemetoder for å kartlegge hvor mye ressurser CORBA-applikasjonen bruker. Ved å sammenlikne med en tilsvarende applikasjon, som ikke benytter CORBA, er det mulig å fastslå hvor mye *overhead* bruken av CORBA medfører.

2.4.5 Kjøremiljø

CORBA-applikasjoner er, som alle andre applikasjoner, avhengig av operativsystemet og de delte maskinressursene. Med andre ord vil implementasjon av operativsystemet spille en avgjørende rolle for ytelsen til en CORBA-basert applikasjon. Andre prosesser som benytter seg av de delte ressursene, vil også påvirke ytelsen til CORBA-applikasjoner, noe som er et reelt scenario for maskiner i "den virkelige verden". Ytelsesmålinger bør derfor gjøres med bakgrunnsbelastning av både nettverk og prosessor for at de skal være reelle[11][10].

2.4.6 Analyse på håndholdte enheter

Håndholdte maskiner er, som beskrevet tidligere, utstyrt med langt mindre prosessorkraft og minne enn de stasjonære maskinene — begrenset av strømkapasitet og fysisk størrelse på enhetene. Ressursforbruket til applikasjonene som kjører på håndholdte enheter er derfor spesielt viktig. Nettverksteknologiene som benyttes på disse maskinene har som regel lang lavere gjennomstrømning enn nettverksteknologiene som benyttes for stasjonære maskiner. Det er derfor ekstra interessant å overvåke forbruk av disse ressursene under ytelsesanalyser av CORBA-implementasjonen, for



Figur 2.8: Oversikt over IEEE 802-familien av nettverksstandarder (akronymene i parentes i det fysiske laget for 802.11 er radioteknologien).

å kunne påvise eventuelle flaskehalser i kombinasjonen av håndholdte enheter og CORBA.

2.5 Nettverksteknologier

Nettverksteknologiene har utviklet seg enormt de siste årene. Standard 10 Mbit Ethernet har blitt erstattet med 100 Mbit. Gigabit nettverk har også vært på markedet i noen år og prisen på slikt utstyr har sunket drastisk. Vi ser nå en kraftig vekst i bruken av de trådløse nettverkene. Stadig flere kontorbygninger og private hjem blir utstyrt med trådløst nettverk i stedet for dyr kabling. På mobilfronten ser vi også en kontinuerlig utvikling av nettverksteknologiene; GSM, UMTS, GPRS og Bluetooth er noen av de mer kjente navnene.

2.5.1 IEEE 802.11b

Institute of Electrical and Electronics Engineers (IEEE) lanserte i 1997 en standard for trådløs kommunikasjon, *IEEE 802.11*. Denne standarden er designet for å være mest mulig lik Ethernet for brukere, og har vokst til å bli en av de mest utbredte teknologiene for trådløse nettverk. Det viste seg raskt at overføringsraten på 2 Mbps var en begrensning, og i 1999 kom derfor to utvidelser av det fysiske laget i den opprinnelige standarden: *802.11a* og *802.11b*. Disse ga mulighet for overføringer på henholdsvis opptil 54 Mbps og 11 Mbps. Sistnevnte er den teknologien som i dag er mest utbredt [13].

IEEE 802.11b er en del av en familie av nettverksstandarder fra IEEE, og er kun en utvidelse av 802.11-standarden på det fysiske laget, hvor radio-

kommunikasjonsteknologien er implementerert. Figur 2.8 gir en oversikt over 802-familien av nettverksstandarder med referanse til de to laveste lagene i OSI-modellen.

802.11b bruker, i likhet med 802.11, 2,4GHz-båndet til kommunikasjon. Dette i motsetning til 802.11a, som bruker 5GHz-båndet. Etterhvert som bruken av 802.11b ble mer og mer utbredt så man behovet for å sørge for interoperabilitet mellom 802.11-produkter. *Wireless Ethernet Compatability Alliance* (WECA) har en sertifiseringsstandard for interoperabilitet mellom 802.11b-produkter kalt *Wi-Fi*, som de fleste slike produkter oppfyller. 802.11b-teknologien blir av denne grunn ofte omtalt som *Wi-Fi*.

802.11b er tiltenkt brukt i såkalte *lokalnettverk* (LAN), og støtter to forskjellige nettverksmodus: *Independent Basic Service Set* (IBSS) og *Infrastructure BSS*, gjerne kalt henholdsvis *ad-hoc*-modus og *managed*-modus. I ad-hoc-modus kommuniserer Wi-Fi-enheter direkte med hverandre, mens all kommunikasjon i en infrastruktur-konfigurasjon går via et *aksesspunkt* (AP). Flere BSS-er i managed-modus kan slås sammen til et *Extended Service Set* (ESS), hvor de respektive AP-ene fungerer som broer mellom nettverkene.

2.5.2 Bluetooth

Bluetooth er en radioteknologi med kort rekkevidde designet for å være billig og bruke lite strøm. Arbeidet med standarden, som har fått sitt navn etter den danske vikinghøvdingen Harald Blåtand, tok til i 1994, da mobiltelefonprodusenten Ericsson begynte å lete etter et alternativ til kabler og IR for å koble telefonene sine til andre enheter. Etterhvert fikk de flere store selskaper med på laget, og i 1998 ble *Bluetooth Special Interest Group* dannet. Litt over ett år senere kom versjon 1.0 av Bluetooth-standarden [14].

Med fremveksten av Bluetooth har det dannet seg et nytt nettverksbegrep, *personlige nettverk* (PAN). Navnet gjenspeiler den bruk Bluetooth er tiltenkt: Kommunikasjon med enheter brukeren omgir seg med i nærheten eller har på seg. Hver slik enhet har sin unike adresse og kan operere enten i *master*- eller *slave*-modus. Flere enheter kan organiseres i nett; da enten i *pico*-nett eller *scatter*-nett. I *pico*-nett har man en enhet som virker som *master* og opptil syv slave-enheter, mens *scatter*-nett består av flere *pico*-nett som er slått sammen.

Pakketype	*Nyttelast*	*Tidsluker*	*CRC*	*FEC*
DM1	17 bytes	1	✓	✓
DH1	27 bytes	1	✓	
DM3	121 bytes	3	✓	✓
DH3	183 bytes	3	✓	
DM5	224 bytes	5	✓	✓
DH5	339 bytes	5	✓	
AUX1	29 bytes	1		

Tabell 2.1: Bluetooth-pakketyper.

Bluetooth opererer i samme frekvensområde som 802.11b. Dette er et lisensfritt frekvensområde, kalt *ISM*-båndet, som støtter tre forskjellige grader av sendestyrke fra 1 mW til 100 mW. De fleste Bluetooth-produkter befinner seg i klasse 1 (1 mW), som gir en rekkevidde på omtrent 10 meter. Til sammenlikning har den kraftigste sendestyrke-klassen en maksimumsrekkevide på rundt 100 meter. Siden man i det ulisenserte ISM-båndet ikke har noen garanti for at ikke-Bluetooth-enheter bruker samme frekvens, benytter man en teknologi kalt *frekvenshopping*. Dette innebærer at etter at en pakke har blitt sendt bytter man frekvens for forbindelsen. 79 kanaler brukes til dette, og det hoppes mellom frekvenser opptil 1600 ganger i sekundet.

Bluetooth-standarden er designet med tanke på både tale- og dataoverføringer. To forskjellige forbindelsestyper er definert for disse bruksområdene: *Synchronous Connection Oriented* (SCO) for tale og lyd, og *Asynchronous Connectionless* (ACL) for data. For ACL-forbindelser har vi syv forskjellige pakketyper for dataoverføringer. Pakkene består av en *Access Code* på 72 bits, en *header* på 54 bits og en nyttelast som varierer for pakketypene. Figur 2.1 gir en oversikt over sentrale egenskaper for disse typene[15]. Alle pakketypene, bortsett fra *AUX1*, støtter 16bits sjekksummer (CRC), og feil i dataoverføringen kan derfor oppdages. *Data Medium-rate*-pakkene (DM), støtter i tillegg *Forward Error Correction* (FEC), og vil selv kunne rette mindre overføringsfeil i nyttelasten. Utover dette er pakketypene delt inn i tre forskjellige pakkestørrelser, definert etter hvor mange tidsluker de opptar.

Ytelsen til Bluetooth i form av overføringskapasitet varier mye avhengig av sendeforhold og hvilke pakketyper man bruker. For asymmetriske[4]

[4]Ved symmetrisk overføring har man samme pakkestørrelse i begge retninger, mens den ved asymmetrisk overføring vil variere.

ACL-forbindelser, hvor man bruker DH5-pakker (*5-slot*, fem tidsluker) i den ene retningen, og *single-slot*-pakker (én tidsluke) i motsatt retning, er maksimal kapasitet i *5-slot*-retningen 732,2 Kbps. Dette er vel å merke på på meget lavt nivå i Bluetooth-protokollen, og man kan ikke regne med å få denne kapasiteten på applikasjonsnivå. For symmetriske overføringer, med 5-slot-pakker i begge retninger, er tilsvarende tall 433,9 Kbps.

De fleste nyere Bluetooth-enheter støtter versjon 1.1 av standarden. Radiogruppen i Bluetooth SIG jobber nå med en version 2.0 av standarden, som vil ha økt overføringskapasitet.

2.5.3 IrDA

Infrared Data Association (IrDA) publiserte i 1993 en pakke med standarder for overføring av data basert på infrarødt lys. Siden den gang har IrDA rukket å bli en utbredt standard for kommunikasjon med en lang rekke IrDA-utstyrte enheter. Teknologien har åpenbare begrensninger i form av kort rekkevidde og behov for uhindret sikt mellom kommuniserende enheter. Enkelte vil derimot hevde at nettopp dette er teknologiens styrke, gjennom å tilby relativt sikker kommunikasjon uten enkle muligheter for avlytting.

Rekkevidder for IrDA-enheter varierer fra 20cm for enheter som opererer i strømsparingsmodus til over 1 meter — gjerne opptil 2 — for enheter i vanlig modus[16]. For såkalt *Serial Infrared* (SIR) er dataoverføringsrater fra 9.600 bps til 115.200 bps, samt 0,576 Mbps, 1,152 Mbps, 4,0 Mbps og 16 Mbps definert.

2.5.4 Ethernet

På midten av 70-tallet utviklet forskere ved Xerox Palo Alto Research Center en nettverksteknologi kalt Ethernet. Teknologien bygger på en mer generell teknologi for lokale nettverk (LAN), *CSMA/CD*[5], og har sitt utspring i et tidlig pakke-radio-nettverk, Aloha, utviklet ved *University of Hawaii*. Denne teknologien var så suksessfull at Xerox, DEC og Intel gikk sammen om å definere en standard for 10 Mbps-Ethernet. Denne dannet senere grunnlaget for dagens IEEE 802.3-standard, som kom i 1983.

[5]Carrier Sense, Multiple Access with Collision Detect

IEEE-standard	*Navn*	*Hastighet*	*Kabeltype*
802.3	10Base-5	10 Mbps	Koaksial (500m strekk)
802.3a	10Base-2	10 Mbps	Koaksial (200m strekk)
802.3i	10Base-T	10 Mbps	UTP kategori 3
802.3u	100Base-T	100 Mbps	UTP kategori 3/5 og fiber
802.3z	1000Base-X	1000 Mbps	Fiber og STP

Tabell 2.2: Utvalg av Ethernet (802.3)-standarder.

I dag finnes det flere varianter av 802.3-standarden. Tabell 2.2 viser noen av de mer kjente for bruk mot arbeidsstasjoner. IEEE 802.3a, også kalt *thin ethernet*, bruker en tynnere og billigere koaksial-kabel enn 802.3, som forenklet kabling av nettverket. I 1990 kom 802.3i-standarden, som gjorde det mulig å bruke en annen type kabel som allerede var i utstrakt bruk i mange kontorbygg, *Unshielded Twisted Pair* (UTP). I tillegg la standarden opp til en stjerneformet nettverkstopologi bestående av punkt-til-punkt-koblinger, noe som gjorde vedlikehold enklere. Disse faktorene førte til en kraftig økning i bruken av Ethernet.

Fem år etter introduksjonen av IEEE 803.2i kom en forbedring av denne standarden. 803.2u, med endret standard for det fysiske laget, muliggjorde en hastighet på 100 Mbps. Denne standarden, også kalt *Fast Ethernet*, støttet både bruk av to par *kategori 5* UTP-kabel (100Base-TX) og fire par *kategori 3* UTP-kabel (100Base-T4), samt to optiske fibre (100Base-FX). Senere har det blitt utviklet en standard for 1 Gbps-nettverk, 1000Base-X, som benytter seg av to forskjellige typer laser over fiber og *Shielded Twisted Pair* [17][18].

2.5.5 USB

Intel lanserte ideen til *Universal Serial Bus* (USB) ut fra et ønske om å standardisere måten perifert utstyr koblet seg opp mot datamaskiner på. Tradisjonelt sett har man hatt mange forskjellige typer kontakter for slikt utstyr, det være seg tastatur, mus, skrivere, eller enheter som har brukt seriell-porten. En del av disse har i tillegg hatt egen strømforsyning. Intel tok sikte på på tilby en type kontakt for strøm og dataoverføring til utstyr uten behov for høyhastighetsoverføring, og i 1995 lanserte de USB 1.0-standarden, godt støttet av andre firmaer. Senere har vi fått en raskere versjon 2.0 av standarden.

USB-enheter har en egen identifikator, og på den måten kan operativsystemet selv gjenkjenne enheter som kobles til datamaskinen og konfigurere systemet. USB-standarden tillater sammenkobling av flere USB-enheter ved bruk av *USB-hub*-er.

Versjon 1.1 av USB-standarden tillater overføringshastigheter opptil 12 Mbps, mens den nyere 2.0-standarden støtter hastigheter opptil 480 Mbps. USB 2.0 er en sterk konkurrent til *FireWire*-standarden og SCSI, som har vært de to aktuelle teknologiene for enheter som har behov for å overføre større mengder data.

2.6 Relatert arbeid

En rekke forskningsprosjekter har tatt for seg forskjellige aspekter ved CORBA. Mest relevant for vårt prosjekt, er ytelsesanalyser av forskjellige CORBA-implementasjoner. Vi har ikke klart å finne prosjekter som er direkte relatert til vårt i form av ytelsesanalyser av CORBA-implementasjoner på håndholdte enheter.

Douglas Schmidt er en sentral person i arbeidet som er gjort med CORBA-implementasjonen *TAO*. Hovedsaklig er dette arbeidet fokusert på optimalisering av CORBA for *real time*-tjenester i høyhastighetsnettverk[9][19][20][21]. Optimalisering med tanke på bruk av høyhastighetsnett er ikke direkte relatert til vårt arbeide med håndholdte enheter, men arbeidet med optimalisering som medfører lavere forbruk av båndbredde og prosessorkraft er definitivt interessant.

Til det modulbaserte måleverktøyet *Netspec*, nærmere omtalt i delkapittel 3.2.4, er det skrevet en egen CORBA-modul som har fått navnet *Performance Measurement Object* (PMO)[22]. Ytelsesmåling av CORBA-implementasjoner kan gjøres parallelt med andre Netspec-moduler, eksempelvis belastning av båndbredde. PMO-utviklerene presenterer også målinger gjort med verktøyet mot OmniORB og TAO[23].

I artikkelen *Performance Evaluation of Object Oriented Middleware*[24] presenteres et måleverktøy for forskjellig typer mellomvare. Forfatterene har utført ytelsesmålinger mot Orbix (C++-versjonen), TAO, Visibroker, DCOM, OrbixWeb, JDK 1.2 og RMI og presenterer et utdrag av disse i artikkelen. Resultatene viser at det er relativt stor forskjell mellom bruk av C++ og Java; mellomvaren implementert i C++ er betydelig raskere.

Forfatterene konkluderer også med at også ytelsen til enkelte av CORBA-implementasjonene er veldig lovende og at *overheaden* mellomvaren påvirker ikke nødvendigvis er stor. En nærmere beskrivelse av måleverktøyet gis i en egen artikkel[25].

Kapittel 3

Design av testoppsett

Dette kapittelet omhandler forarbeidet til ytelsesanalysene. Vi gir først en beskrivelse av hva vi definerer som hovedmål og delmål for oppgaven, og redegjør deretter for våre ressursbehov — hva vi anser som nødvendig av ressurser for å nå målet. Vi avslutter med å beskrive hvilke test-scenarier vi ser for oss, og hvilke typer testkonfigurasjoner som lar oss gjøre målinger på disse.

3.1 Overordnede krav

Vårt hovedmål er å undersøke om håndholdte enheter er i stand til å kjøre ressurskrevende mellomvare som CORBA. Vi ønsker å se på hvilke begrensninger ressurssituasjonen på de tynne klientene eventuelt legger på bruken av en CORBA-implementasjon; vil det for eksempel kun være mulig å bruke håndholdte enheter som klienter mot CORBA-tjenester, eller vil de også kunne fungere som tjenere?

For å nå det overordnede målet har det vært nødvendig å se nærmere på forskjellige aspekter ved en ORB. Vi ønsket å måle *responstid*, *gjennomstrømning*, *skalerbarhet* og *ressursforbruk*. Dette er de samme aspektene som beskrives i OMGs artikkel om ytelsesmålinger av CORBA-implementasjoner [10], og tilsammen gir de et bilde av hvor høy ytelse en CORBA-implementasjon kan gi på en bestemt plattform.

Under analyse av data fra målingene er det viktig å se på hvilke faktorer som har påvirkning på resultatet. Vi har derfor valgt å overvåke ressurs-

forbruk i form av minne, prosessor og båndbredde under ytelsesmålingene. Dette er spesielt viktig på håndholdte enheter, hvor disse ressursene er langt mer begrenset enn på arbeidsstasjoner.

De fleste av dagens operativsystemer håndterer flere kjørende prosesser, og fordeler tilgjengelige maskinvare-ressurser mellom dem. Under ytelsesmålinger vil en testapplikasjon av den grunn ikke nødvendigvis ha eksklusiv tilgang på systemressurser — en kan ikke utelukke at andre prosesser kjører i bakgrunnen og dermed legger beslag på ressurser (bakgrunnslast). Dette vil kunne påvirke måleresultater, og vi har derfor ansett det som viktig å måle denne bakgrunnsbelastningen.

Hvordan ressurser blir fordelt vil variere fra operativsystem til operativsystem. I tillegg vil det være forskjeller i hvor effektivt minnehåndtering, protokoller, og andre faktorer som virker inn på ytelse, er implementert. Valg av maskinvare og operativsystem vil av denne grunn få innvirkning på resultatene av ytelsesmålingene, og vi planla derfor å kjøre målinger på forskjellige kombinasjoner av operativsystem og maskinvare. Ved å gjøre dette ville vi se hvilke potensielle ytelsesforskjeller plattformvalget gir.

Valg av nettverk for kommunikasjon mellom klient- og tjener-ORB vil kunne påvirke ytelsen til en ORB. Egenskaper ved en nettverksteknologi, som latens og båndbredde, kan ha en stor innvirkning på resultatet av målingene. Ved bruk av trådløse nettverk sammen med mobile enheter er det nærliggende å anta at kvaliteten på nettverksforbindelsen vil variere, og i enkelte tilfeller kan også forbindelsen falle helt bort. Vi har imidlertid valgt å ikke se nærmere på mobilitetsaspektet, men kun gjøre målinger under tilnærmet optimale nettverksforhold.

I følge Jim Gray[12] skal benchmark-algoritmer være portable, slik at de kan benyttes i forskjellige miljøer med forskjellige arkitekturer. Ved å bruke de samme algoritmene som andre har brukt til ytelsesmåling vil man få et større sammenlikningsgrunnlag ved analyse av egne data. Det er derfor gunstig å benytte et ferdigutviklet verktøy for målinger på forskjellige plattformer.

3.2 Behov for maskin- og programvare

For å nå målene vi har satt oss, har vi behov for en både maskin- og programvare. Av maskinvare trengs det håndholdte og stasjonære maskiner

Produsent	*Modell*	Lansert*OS*	*CPU*	*RAM*	*ROM*
Palm Inc.	Palm V	PalmOS	16 MHz	8MB	2MB
Palm Inc.	Palm i705	PalmOS	33 MHz	8MB	4MB
Handspring	Treo 90	PalmOS	33 MHz	16MB	N/A
Sony	Clie PEG-T665C	PalmOS	66 MHz	16MB	8MB
Audiovox	Maestro PDA-1032	Windows CE	206 MHz	32MB	32MB
NEC	MobilePro P300	Windows CE	206 MHz	32MB	32MB
Compaq	iPAQ H3870	Windows CE	206 MHz	64MB	32MB
HP	iPAQ H3950	Windows CE	400 MHz	64MB	32MB
Sharp	Zaurus SL-5500	Linux	206 MHz	64MB	16MB

Tabell 3.1: Utvalg av håndholdte enheter i 2002. Palm V ble lansert i 1999 og er ikke lenger i produksjon.

med nettverkskort for kommunikasjon med andre enheter.

Av programvare er det først og fremst behov for CORBA-implementasjoner, som kan kjøres på både håndholdte og stasjonære maskiner. Videre trenger vi et analyseverktøy som kan måle ORB-enes ytelse, samt registrere hvor mye systemressurser som har blitt lagt beslag på.

I dette delkapittelet beskriver vi kandidater innen maskinvare, nettverksutstyr, operativsystemer, analyseverktøy og CORBA-implementasjoner, som vi har vurdert for bruk i testkonfigurasjonen. Til slutt oppsummerer vi med en evaluering, samt en oversikt over våre valg innen de respektive kategoriene.

3.2.1 Maskiner

For å lette utviklingen av programvare for håndholdte enheter, er det vanlig å benytte seg av programvare som emulerer maskinvaren. En kan da utvikle og teste programmer på en stasjonær maskin. *PalmOS Emulator* (POSE) er eksempel på en slik emulator. POSE kan brukes til å teste applikasjoner uten å ha en Palm, og det er i teorien mulighet for å kjøre ytelsestester på den. For å oppnå realistiske ytelsesmålinger har vi imidlertid behov for reell maskinvare, ettersom en emulator ikke kan emulere maskinvare og nettverksforbindelser godt nok. Tabell 3.1 gir en oversikt noen

Operativsystem	*Antall ulike modeller*
PalmOS	43
Windows CE	33
Linux	1
Symbian	1
Andre (4)	10

Tabell 3.2: Fordeling av operativsystemer på håndholdte maskiner (basert på en oversikt fra CNet)

av de håndholdte enhetene som er på markedet i 2002, og hvordan de er utstyrt.

Utelukkende bruk av håndholdte enheter under ytelsesmålingene vil ikke gjenspeile realistisk bruk, da slike enheter gjerne også kommuniserer med kraftigere arbeidsstasjoner. Vi har derfor, i tillegg til målinger mellom håndholdte enheter, også ønsket å gjennomføre målinger mellom håndholdt og stasjonær maskin. I det følgende vil vi beskrive ulik maskinvare vi vurderte som aktuelle til slike målinger.

Palm V

Palm V ble lansert i begynnelsen av 1999, og er utstyrt med en 16MHz Motorolla Dragonball-prosessor og 2MB RAM. Videre er den utstyrt med en kommunikasjonskontakt for tilkobling av ekstrautstyr. Ved å benytte seg av medfølgende *dockingstasjon* kan man via denne kontakten kommunisere over en seriell linje mot en arbeidsstasjon. Palm V har også innebygget IrDA-støtte, som muliggjør kommunikasjon mot andre IrDA-enheter.

Som en del av MULTE-prosjektet kjøpte vi inn et antall Palm V for å studere mellomvare på håndholdte enheter. Disse ble levert med ekstra minne, 8MB mot standard 2MB, for å kunne kjøre mer ressurskrevende programmer. Palm V ble levert med operativsystemet PalmOS 3.0, men vi oppgraderte dem senere til PalmOS 4.0. Enheten kan også kjøre en spesialtilpasset versjon av Linux, μClinux, som beskrives nærmere i delkapittel 3.2.3.

iPAQ H3870

iPAQ H3870 er utstyrt med en 206MHz Intel SA-1110 StrongARM prosessor, 64MB RAM og 32MB ROM, og befinner seg i klassen for de raskeste håndholdte maskinene. Enheten er utstyrt med serieport, USB, IrDA og Bluetooth. Den kan også brukes sammen med en *PC Card sleeve*, som man trer iPAQ-en ned i. Denne har støtte for *PC Card*-kort[1], og man kan således benytte nettverkskort for trådløs kommunikasjon med iPAQ. H3870 leveres med operativsystemet Windows CE, men også denne enheten kan kjøre Linux.

Uninetts UMNS-prosjekt har valgt å satse på iPAQ som maskinvare-plattform. Vi har gjennom dette prosjektet fått tilgang på to stk H3870 med nødvendig nettverksutstyr. Hewlett Packard[2] har, etter at disse var innkjøpt, lansert en kraftigere iPAQ, H3970, utstyrt med en 400 MHz-prosessor basert på Intels nye *XScale*-arkitektur.

Arbeidsstasjon

Ettersom CORBA-implementasjoner og programvare for ytelsesmålinger som regel er utviklet for bruk på vanlige arbeidsstasjoner, fant vi det praktisk med en arbeidsstasjon til innledende utprøving av både CORBA-implementasjoner og programvare for ytelsesmålinger. Arbeidsstasjonen kan i tillegg benyttes til testing av CORBA-implementasjoner mot en håndholdt enhet.

Arbeidsstasjonen vår er utstyrt med en Intel Pentium II 350MHz prosessor og 512MB RAM. I tillegg har vi, for å kunne gjøre målinger over IrDA, i en periode benyttet en arbeidsstasjon med innebygget IR-port. Denne er utstyrt med 500MHz Pentium III prosessor og 384MB RAM.

3.2.2 Nettverk

For håndholdte enheter er det mest naturlig å se for seg CORBA brukt i en distribuert sammenheng, og vi har derfor fokusert på ytelsesmålinger gjort mellom CORBA-klient og tjener over en nettverksforbindelse. Flere nettverkstyper er aktuelle for håndholdte: Ethernet, 802.11b, Bluetooth,

[1] Kort av type 1 og 2 støttes, mens CardBus ikke er støttet.
[2] Compaq ble kjøpt av Hewlett Packard mai 2002.

IrDA og USB. For de tre sistnevnte, med unntak av IrDA i FIR-modus, brukes en PPP-forbindelse slik at man kan bruke TCP/IP-sockets.

De nevnte nettverksteknologiene har alle forskjellig båndbredde, og det er nærliggende å anta at dette vil virke inn på resultatet av ytelsesmålingene. Vi anser det som interessant å sammenlikne resultater fra målinger gjort over flere forskjellige typer nettverk, og benytter oss derfor av flere ulike nettverksteknologier i målingene. Vi gir her en oversikt over nettverksprodukter vi vurderte.

Wi-Fi (802.11b)

D-links *DI-713P Wireless Broadband Router* er en 3-ports 10BASE-T/100BASE-TX-ruter med 802.11b-støtte. Ruteren har mulighet for filtrering på MAC-adresser[3], og det lar seg dermed gjøre å begrense Wi-Fi-trafikken til kun de maskinene involvert i testing.

Cisco Aironet 350 Wireless Lan Adapter er et 802.11b PC Card-kort. Kortet er støttet av Linux, både på i386- og StrongARM-plattformen. Cisco leverer *Open Source*-drivere til dette kortet, som støtter både *Managed*- og *Ad-Hoc*-modus.

Bluetooth

3Coms *Wireless Bluetooth USB Adapter* muliggjør målinger over Bluetooth mellom iPAQ og arbeidsstasjon. Adapteret støtter Bluetooth 1.1-standarden, 128bit-kryptering og sendestyrken er i *klasse 2*, noe som innebærer at rekkevidden er omtrent 10 meter.

TDK blue5 Bluetooth enabler er et adapter som kobles på Palm V(x), og gir Palm-enheten Bluetooth 1.0b-støtte. Den befinner seg i samme styrkeklasse som 3Coms adapter.

IrDA

Både Palm V, iPAQ H3870 og den ene arbeidsstasjonen har innebygget IrDA-port. Palm V støtter kun *Serial IR* (SIR), mens iPAQ-en og arbeids-

[3]En MAC-adresse er en maskinvare-adresse som er unik for hvert produserte nettverkskort

stasjonen i tillegg støtter *Fast IR* (FIR). Imidlertid er FIR-støtten for brikkesettet (SMCF010) brukt i arbeidsstasjonen noe ustabil under Linux.

Ethernet (802.3u)

Et 100Mbit Ethernet-kort til iPAQ vil la oss gjøre målinger over en nettverksforbindelse med betydelig større båndbredde enn de andre nettverksteknologiene. Imidlertid er det svært få nettverkskort av PCMCIA/PCCard-typen som er støttet under Linux/ARM, og vi har dessverre ikke klart å anskaffe et slikt.

USB

iPAQ-ene er utstyrt med en *USB-vugge* som støtter USB 1.1-standarden. I tillegg er det mulig å bruke en spesialtilpasset USB-kabel.

3.2.3 Operativsystemer

Operativsystemet har innvirkning på ytelsen til programmene som kjører på maskinen. Prosess- og minnehåndtering, samt nettverksimplementasjon, er alle faktorer som kan gi merkbare utslag i ytelsen. Det vil derfor være interessant å gjøre målinger ved bruk av flere forskjellige operativsystemer for å kartlegge denne innvirkningen.

PalmOS

Utover Palms egne enheter leveres en rekke håndholdte enheter fra forskjellige produsenter med PalmOS. Operativsystemet er kompakt, krever lite ressurser, og er skreddersydd for ressurssvake enheter. PalmOS støtter ikke multithreading, og dette gjør at operativsystemet er mest aktuelt som klient i CORBA-sammenheng. Om en tjener-prosess kjøres på maskinen, vil man ikke kunne bruke enheten til andre oppgaver, og det vil være tilnærmet umulig å betjene flere klienter samtidig[20]. I tillegg har PalmOS en begrensning på maksimalt fire simultant åpne TCP-sockets. Dette skal ha blitt gjort for å spare minne, men vil dessverre også virke som en begrensning ved bruk av CORBA.

Windows CE

Microsoft Windows CE er et kompakt, flertrådet operativsystem utviklet for integrerte enheter. OS-et modulært oppbygget, med en liten kjerne og et sett med moduler som tilbyr ulik funksjonalitet. De mest sentrale modulene gir støtte for filsystem, grafikk og kommunikasjon, men også ytterligere moduler for andre behov eksisterer.

For utviklere på denne plattformen har Windows CE et *Application Programming Interface* (API), som er basert på Windows NTs *WIN32*-API. API-et for CE er imidlertid noe redusert i forhold til NTs for å spare plass, men tilbyr til gjengjeld funksjonalitet tilpasset bruk på integrerte enheter. For kommunikasjon støttes blant annet TCP/IP, IrDA og PPP[26][27].

Linux

Linux er et operativsystem basert på frivillig innsats og åpen kildekode. Operativsystemet kan brukes på en rekke arkitekturer, inkludert Intel x86 og StrongARM, samt Motorola 68000-serien. Fordelen med Linux er med andre ord at det kan brukes på alle maskinarkitekturene vi har sett på.

Linux/Microcontroller-prosjektet *μClinux* har laget en Linux-kjerne tilpasset maskiner uten minnehåndteringsenhet (MMU). Den er basert på en Linux 2.0-kjerne, men er fullstendig omskrevet for å minimere størrelsen; resultatet er en kjerne som er langt mindre enn den opprinnelige 2.0-kjernen. Deres første versjon har vært for *Motorola Dragonball*-prosessoren, som Palm bruker i sine enheter. Bluetooth er imidlertid ikke støttet i Linux 2.0-kjernene, og det er derfor ikke mulig å bruke denne nettverksteknologien på Palm med μClinux.

μClinux er blant annet benyttet i en kommersiell Linux-distribusjon for Palm, *LinuxDA*. LinuxDA er en komplett pakke med operativsystemkjerne, grafisk grensesnitt og programmer tilsvarende de som følger med PalmOS.

Også for StrongARM-arkitekturen finnes det en egen versjon av Linux. Denne er inkludert i kildekode-treet til Linux og støtter SA-1110-prosessoren som benyttes på de fleste Windows CE-baserte enhetene, inkludert iPAQ (se delkapittel 3.1). Basert på denne vedlikeholdes en iPAQ-tilpasset versjon med støtte for maskinvaren — hovedsaklig touchscreen, lysmåler, mikrofon — som finnes i disse enhetene.

Forskjellige Linux-distribusjoner eksisterer for håndholdte basert på StrongARM-arkitekturen: *ELinOS*, *MontaVista Linux*, Mizis *Linu@*, Lineos *Embedix Plus PDA*, *LEM*, PalmPalms *Tynux*, *Familiar Linux*, og *Intimate Linux*. Av disse er kun de to sistnevnte ikke-kommersielle. Familiar er basert på *Debians* ARM-distribusjon, og er tilpasset installasjon i ROM på iPAQ, mens *Intimate* er en langt mer omfattende pakke som krever mere lagringsplass i form av ekstra minnekort, harddisk-kort eller liknende.

3.2.4 Analyseverktøy

Utvikling av et analyseverktøy, som skissert i OMGs *White Paper on Benchmarking* [10], er en omfattende oppgave. I stedet for å bruke tid på utviklingen av et eget verktøy, har vi valgt å bruke eksisterende analyseverktøy.

Netspec

Netspec er et modularisert analyseverktøy for måling av nettverksytelse[28]. Verktøyet består av flere moduler som startes som demoner ved hjelp av en skriptstyrt kontrollerdemon. Denne løsningen gjør det mulig å sette opp komplekse ytelsesmålinger hvor maskiner og testmoduler varierer. Eksempelvis kan Netspec brukes for å måle TCP- og UDP-ytelsen over et nettverk. I tillegg kan analyseverktøyet brukes til å emulere forskjellig trafikk, som for eksempel HTTP, FTP, Telnet og CORBA.

For å gjøre CORBA-målinger med Netspec har *Team Niehaus*[4] skrevet en CORBA-modul. Modulen benytter seg av *Performance Measurement Objects* (PMO)[29][22], hvor objektene kalles med parametre og utfører operasjoner på de overførte dataene. Deretter samles ytelsesdataene og sendes tilbake til kontrolleren. Team Niehaus har skrevet PMO for TAO, OmniORB2 og ILU.

Av det vi har funnet av informasjon og kildekode, ser det ut til at både Netspec og CORBA-modulen sist ble oppdatert i 1997.

[4]Team Niehaus er en gruppe med studenter som jobber for Dr. Douglas Niehaus, University of Kansas.

Open CORBA Benchmarking

Open CORBA Benchmarking (OCB) er et prosjekt som har som mål å lage et åpent analyseverktøy som dekker all "vanlig" bruk av en CORBA-implementasjon. Slike verktøy blir ofte store og uhåndterlige, og forfatterne har derfor delt inn potensielle brukere i to grupper etter hvilket behov de vil ha: ORB-utviklere og ORB-brukere. Det er foreløpig kun implementert et analyseverktøy for sistnevnte gruppe.

Ytelsesmålingene har de delt inn i flere separate tester: responstid, gjennomstrømming, skalerbarhet og parallellitet, samt en test hvor disse kombineres. Verktøyet inkluderer også måling av systemytelse (*microbenchmarks*[30]). Denne målingen er ment å gi perspektiv på måleresultater, slik at man lettere kan forutsi ressursbehov ved å gjøre sammenlikninger med resultater fra andres målinger. For å bygge opp et sammenlikningsgrunnlag OCB-utviklerene opprettet en sentral database, hvor resultater kan registreres.

Prosjektet baserer seg på åpen kildekode og samler måleresultatene i en XML-datafil. Denne filen registreres så i databasen, og analysen blir så tilgjengelig via et nettsted. Her kan man se resultatet fra de ulike testene, samt se tall på usikkerheten i målingene. Testverktøyet er allerede tilpasset MICO, omniORB, ORBacus, ORBacus/E, Orbix 2000, Orbix/E, TAO og VisiBroker.

Ressursforbruk

For å analysere ressursforbruket under målingene har vi sett nærmere på to verktøy som det ofte blir refert til på websider og nyhetsgrupper: *Ethereal* og *Linux Trace Toolkit*.

Ethereal er et analyseverktøy for nettverksprotokoller på Unix og Windows. Verktøyet logger trafikk på nettverksgrensesnitt, og kan lagre alt som sendes og mottas, slik at man i ettertid kan analysere datatrafikken på pakkenivå.

Linux Trace Toolkit er et verktøy for å overvåke bruk av Linux-kjernen under kjøring. Med verktøyet kan man analysere hvilke deler av kjernen som brukes under kjøringen av et program. I CORBA-sammenheng kan det for eksempel være interessant å vite hvor mye

ORB	*Produsent*
MICO	MICO
OmniORB	AT&T
Orbix 2000	IONA
ORBacus	IONA
Orbix/E	IONA
TAO	Washington Univ. St. Louis
UIC-CORBA	UBI-Core
Visibroker	Borland

Tabell 3.3: CORBA-implementasjoner støttet av Open CORBA Benchmarking.

tid som brukes i kjernens TCP/IP-implementasjon, ettersom IIOP ligger over disse lagene.

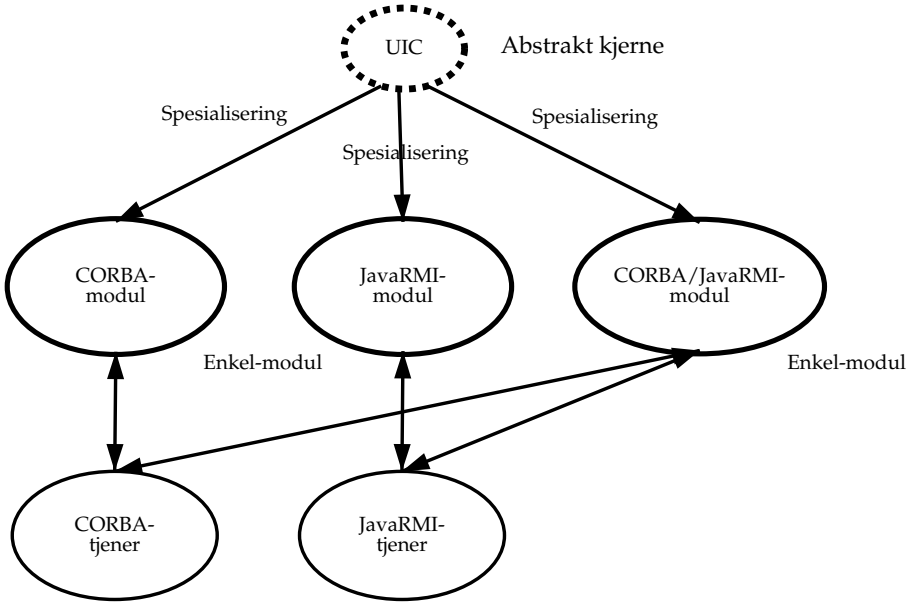
3.2.5 CORBA-implementasjoner

Det finnes et stort utvalg CORBA-implementasjoner på markedet i dag, både gratis og kommersielle. Tabell 3.3 viser et utvalg, som er støttet av måleverktøyet *Open Corba Benchmarking*. Vi har sett på to av disse, *MICO* og *Orbix/E*, samt en tredje, UIC-CORBA.

UIC-CORBA

Universally Interoperable Core (UIC) er en generell ORB-implementasjon, utviklet med tanke på håndholdte enheter[31]. Kjernen er abstrakt og utvides med moduler for å oppnå ønsket funksjonalitet. Flere moduler kan benyttes samtidig, og eksempler på slike er CORBA, Java/RMI og HTTP (se figur 3.1). UIC er skrevet for platformuavhengighet og er testet på Windows, Windows CE og PalmOS. Utviklerene av UIC, *UBI-Core*, har skrevet en CORBA-modul, *UIC-CORBA*, som vi valgte å se nærmere på.

UIC-CORBA tar ikke sikte på å være en full implementasjon av CORBA, men implementerer et subsett av standarden, hvor funksjonalitet det normalt ikke er bruk for på håndholdte enheter er fjernet. Klientimplementasjonen tilbyr funksjonalitet for å generere CORBA-objekter fra



Figur 3.1: Oppbyggingen av UIC [31].

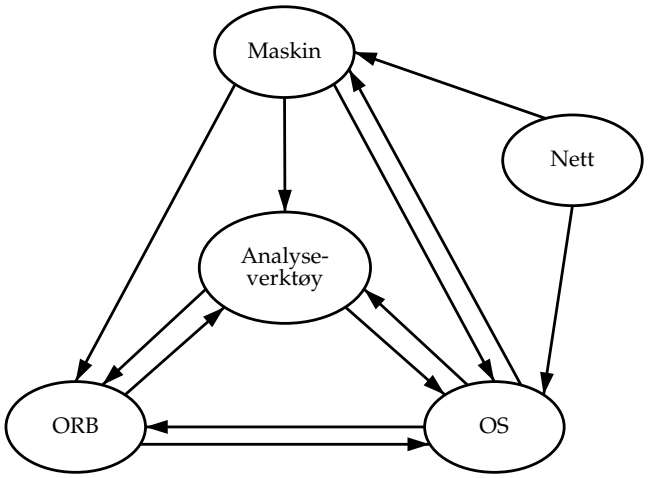
IOR-referanser, et *Dynamic Invocation Interface* (DII), og et *kjerne*-subsett av CORBA-typer, mens tjener-delen tilbyr registrering av tjener-objekter[32]. For å kunne utføre målinger under Linux ble vi nødt til å *porte* UIC-CORBA til dette operativsystemet.

Orbix/E

IONAs *Orbix/E* er en flertrådet lettvekts-ORB som er designet for *integrerte systemer* på håndholdte og mobile enheter. Orbix/E 2.1 er basert på et subsett av CORBA 2.3, og finnes i både en *Java*- og *C/C++*-versjon. Ved å fjerne funksjonalitet som gjerne er lite brukt på integrerte systemer har IONA laget en ORB som krever lite plass. Implementasjonen har full IDL-støtte, med unntak av tre datatyper, og en del feiltyper. CORBAs dynamiske *Any*-type, DII og *Dynamic Skeleton Interface* (DSI) er ikke støttet, og ORB-ens *Portable Object Adapter* er også noe redusert.

Orbix/E er kommersiell, men leveres med kildekode for å være mest mulig fleksibel for brukerene. Java-versjonen er skrevet for J2ME[5] og kan brukes på PalmOS, men kun som klient grunnet den begrensede tråd-støtten i operativsystemet [33]. For Linux er C/C++-versjonen mest aktuell.

[5]Java2 Micro Edition, se `http://java.sun.com/j2me/>`



Figur 3.2: Avhengigheter ved valg av verktøy. Pilene angir avhengighetene.

MICO

MICO er en *Open Source* CORBA-implementasjon implementert i standard C++. MICO-utviklerene har som mål at ORB-en hele tiden følger siste versjon av CORBA-standarden, og nyeste versjon (2.3.7) støtter CORBA 2.3. Videre har de basert seg på standard UNIX API-er, og unngått å implementere funksjonalitet som ikke inngår i CORBA-standarden.

MICO er ikke flertrådet, men finnes også i en versjon, *MICO/MT*, som støtter dette. Videre finnes prosjektet *MICO for the PalmPilot*, som har som målsetning å få *portet* MICO til PalmOS. Prosjektsidene er imidlertid utdaterte, og det virker som om prosjektet har stanset opp.

3.2.6 Valg av maskin- og programvare

Alle valgene vi har gjort er avhengige av hverandre, noe som illustreres av figur 3.2. Valg av maskin har satt krav til hvilket operativsystem, ORB og analyseverktøy som velges. Ved valg av operativsystem har vi vært avhengig av å ha en maskin som det kan kjøres på, i tillegg til at ORB-en og analyseverktøyet må fungere på plattformen. Valg av nettverk har vært det enkleste, ettersom dette kun er avhengig av at vi har riktig maskinvare, samt et operativsystem nettverket kan brukes med.

Kategori	*Valgt kandidat*
Maskiner	iPAQ H3870, arbeidsstasjon
Nettverk	DI-713P, Aironet, 3Com BT, IrDA
Operativsystemer	Linux/x86, Linux/ARM
Analyseverktøy	OCB, egne skript
CORBA-implementasjoner	MICO, Orbix/E

Tabell 3.4: Valg av ressurser

Etter å ha vurdert alternativene for maskin- og programvare, forkastet vi noen grunnet svakheter eller mangler, som ville ha begrenset muligheten til grundige målinger. Andre fant vi tilstrekkelige til å bruke til videre analyse, og vi gir her en kort redegjørelse for våre valg (tabell 3.4 oppsummerer disse).

Maskiner

Med maksimum fire åpne TCP/IP-forbindelser og manglende multithreading innså vi at Palm med PalmOS ikke ville kunne kjøre målinger av skaleringsevne for flere simultane klienter. Vi antok at andre typer målinger ville være mulig å gjennomføre, men etter å ha forsøkt å kjøre *Open CORBA Benchmarking*-testpakken på iPAQ, og sett hva den krevde av systemressurser, fant vi at Palm Vs begrensede minne og svake prosessor ville ha problemer med å kjøre en omfattende analyse. Imidlertid ville det nok ha vært mulig å foreta enkle målinger med Orbix/E som klient på Palm med PalmOS, men dette ville krevd spesialtilpasset analyseverktøy.

Vi vurderte også μClinux som et alternativ til PalmOS. Denne operativsystemkjernen støtter multithreading, og har heller ingen øvre grense for antall åpne sockets slik PalmOS har. μClinux er imidlertid basert på en Linux 2.0-kjerne, og mangler Bluetooth-stack. Målinger over denne nettverksteknologien ville derfor ikke vært mulig. De samme begrensningene i minne og prosessorkraft på Palm V gjelder også med dette operativsystemet, og vi forkastet derfor dette alternativet.

Etter innledende testing av *Open CORBA Benchmarking* under Linux på iPAQ fant vi at iPAQ H3870 er en interessant arkitektur å basere seg på. Med to fungerende operativsystemer, *Linux* og *Windows CE*, og flere tilgjengelige nettverkstyper ville det være mulig å foreta flere målinger med

forskjellige konfigurasjoner. iPAQ ble derfor valgt som maskinvareplattform.

Nettverk

Av Wi-Fi-utstyr virket både *D-Link*-ruteren og *Aironet*-kortene uten problemer. Sistnevnte fikk vi til å fungere på både Linux og Windows på iPAQ umiddelbart, og vi valgte å bruke dette utstyret til våre analyser. Bluetooth-adapteret til 3Com fungerte utmerket under Linux med arbeidsstasjonen vi hadde valgt, mens TDK-adapteret for Palm ble forkastet sammen med Palm V.

Av IrDA ønsket vi å bruke både SIR- og FIR-modus mellom iPAQ-er, og mellom en iPAQ og en arbeidsstasjon. Vi fikk imidlertid problemer med FIR-modus for arbeidsstasjonen, da støtten for IrDA-brikkesettet var mangelfull i Linux. IrDA ble likevel planlagt brukt i analysen.

Etter flere forsøk på test-kjøringer av en benchmark mellom en iPAQ og arbeidsstasjon over USB ble vi nødt til å forkaste USB. Når mengden data som ble overført nådde den maksimale kapasiteten feilet forbindelsen, og selv etter mye feilsøking klarte vi ikke å lokalisere feilen. Mangel på maskinvare gjorde målinger over Ethernet umulig.

Operativsystemer

I forbindelse med at vi forkastet Palm som arkitektur forsvant også muligheten for å gjøre målinger på PalmOS. Windows CE ble betraktet som et potensielt OS for å kjøre målinger på, men mangel på tid til å tilpasse måleverktøy skrevet for Linux gjorde at vi så oss nødt til å forkaste dette alternativet.

Etter søking på nettsider og nyhetsgrupper fikk vi inntrykk av at Familiar-distribusjonen var riktig valg av Linux-plattform. Familiar er rettet spesifikt mot iPAQ, er således svært egnet til vårt bruk. Videre er det et meget aktivt utvikler- og bruker-miljø knyttet til denne distribusjonen. Dette gjør at det er store muligheter for å få hjelp om man står fast. Familiar ble derfor installert på én av iPAQ-ene, og etter testing av støtte for de ulike nettverkstypene, samt test-kjøring av *Open CORBA Benchmarking*, valgte vi denne distribusjonen.

Analyseverktøy

Grunnet store problemer med å få Netspec til å fungere, forkastet vi dette verktøyet. Dokumentasjonen var mangelfull, og programvaren lot seg ikke kompilere på nyere Linux-distribusjoner. I tillegg var ikke CORBA-modulen helt ferdig, noe vi fikk vite da vi tok kontakt med Douglas Niehaus, veilederen for utviklerne av Netspecs CORBA-demon.

Open CORBA Benchmarking er et fleksibelt verktøy, og følger retningslinjene for CORBA-ytelsesanalyser som [10] omtaler. OCB er allerede tilpasset flere forskjellige CORBA-implementasjoner, og vi hadde ingen problemer med under test-kjøringer av verktøyet med MICO og Orbix/E. OCB-prosjektet har dessuten en database med målinger, som gir oss tilgang til data for sammenlikning. Disse egenskapene gjorde OCB til et attraktivt valg, og vi bestemte oss for å bruke dette.

Linux Trace Toolkit (LTT) ble forkastet ettersom verktøyet er litt for omfattende for vårt bruk. Det krever en spesialkompilert Linux-kjerne, og vi ønsket ikke å bruke tid på tilpasning av denne for iPAQ. Ved nærmere studier av hva det brukes tid på i operativsystemkjernen ved en CORBA-analyse, ville LTT vært interessant. En slik analyse ligger imidlertid utenfor denne oppgavens rammer.

Ethereal ble forkastet grunnet manglende støtte for ARM-arkitekturen, samt at den ikke tilbyr den type funksjonalitet vi ønsket. Vi hadde ikke behov for å analysere de enkelte pakkene, men ønsket kun statistikk for båndbreddeforbruket til de enkelte nettverks-grensesnittene. I mangel av ferdigskrevne verktøy valgte vi å skrive egne verktøy for dette, samt overvåking av systemressurser.

CORBA-implementasjoner

Porting av UIC-CORBA til Linux tok mye tid, men bød ikke på noen store problemer. Vi klarte også å lage enkle fungerende CORBA-applikasjoner med bruk av ORB-en, og de store problemene kom først da vi prøvde å tilpasse *Open CORBA Benchmarking* til bruk med UIC-CORBA.

OCB baserer seg på bruk av CORBA-implementasjonens IDL-kompilator. IDL-kompilatoren til UIC-CORBA har store mangler og støtter blant annet ikke *moduler*, som OCB benytter seg av. Med hjelp fra UIC-utviklerne skrev vi om OCBs IDL-fil slik at den kompilerte, men koden som ble generert

fulgte dessverre ikke CORBA-standarden, og virket derfor ikke med OCB. Heller ikke assistanse fra en av OCB-utviklerene, Petr Tuma, hjalp. I følge ham var ikke UIC-CORBA i nærheten av å følge CORBA-spesifikasjonen. Vi valgte derfor å forkaste UIC-CORBA.

Orbix/E fungerte uten problemer med Linux, både på arbeidsstasjon og iPAQ. Tilpassing av OCB gikk også smertefritt ettersom OCB allerede var klargjort for en tidligere versjon av Orbix/E (2.0). Kun små tilpasninger var nødvendig. MICO skapte heller ingen problemer ved testkjøringer under Linux på arbeidsstasjon og iPAQ. På lik linje med Orbix/E er også MICO ferdig konfigurert i OCB. MICO/MT, derimot, fikk vi aldri til å fungere, og vi endte opp med å velge Orbix/E og MICO.

3.3 Testoppsett

Vår testkonfigurasjon tar utgangspunkt i de mål vi har satt oss for ytelsesanalysene. I vårt tilfelle ønsket vi å teste forskjellige CORBA-implementasjoner på ulike plattformer og nettverk. Ved å sammenlikne måledata kan vi se hvordan endringer i testparametre som maskinvare og nettverksteknologi påvirker ytelsen til implementasjonene, og hvordan ytelsen varierer for de forskjellige ORB-ene ved ellers like konfigurasjonsparametre.

For å isolere faktorer med innvirkning har vi derfor hatt behov for et oppsett som gir oss mulighet til å variere testparametrene uten at andre deler av testkonfigurasjonen blir endret.

3.3.1 Test-scenarier

Vi ville studere kombinasjoner av klient og tjener, hvor de kjører på maskiner med henholdsvis lik og ulik arkitektur. I tillegg ville vi gjøre målinger hvor klient og tjener kjører på samme maskin. Følgende liste gir en oversikt over forskjellige konfigurasjoner:

1. Klient på iPAQ, tjener på PC
2. Klient på iPAQ, tjener på iPAQ nr. 2
3. Klient på PC, tjener på iPAQ

4. Klient og tjener på samme iPAQ/PC

Det første test-scenariet vi valgte å studere er om man på en iPAQ kan kjøre en CORBA-klient som kommuniserer med en tjener på en arbeidsstasjon. Som scenario nummer to byttet vi ut arbeidsstasjonen med en iPAQ på tjener-siden for å se om det påvirker ytelsen og ressursforbruket på klient-siden. Disse to scenariene lar oss studere hvordan CORBA-tjeneren oppfører seg på to forskjellige plattformer med samme type klient i den andre enden.

Et tredje scenario det er naturlig å se på, for å komplettere de to første, er tjener på iPAQ og klient på arbeidsstasjon. Dette oppsettet, som er "speilvendt" av det første, kan settes opp mot scenario to for å undersøke om bytte av klient-plattform påvirker iPAQ-tjeneren. Med disse tre scenariene ville vi få undersøkt alle klient-tjener-kombinasjoner for iPAQ mot både en arbeidstasjon og en annen iPAQ.

For å få litt perspektiv på målinger gjort under de ovennevnte konfigurasjonene er det interessant med et fjerde scenario: Målinger hvor både klient og tjener kjører på samme maskin. I dette tilfellet vil klient og tjener dele de samme maskinressursene, men kommunisere uten noe nettverkslag — kun socket mot socket.

For hver av forskjellige klient-tjener-konfigurasjonene ønsket vi å studere effekten ved bruk av ulike nettverksteknologier. Wi-Fi, Bluetooth og IrDA er naturlig å se på for håndholdte enheter, men det er også interessent å sammenlikne målinger gjort over samme nettverksteknologier med målinger fra et kraftigere 100 Mbit Ethernet.

I tråd med det overordnede målet med analysen — å finne ut i hvilken grad man kan benytte CORBA på håndholdte — planla vi å benytte ulike CORBA-implementasjoner i disse scenariene for å se om optimalisering for bruk på håndholdte gir merkbare utslag i ytelse. Bruk av forskjellige nettverksteknologier under disse målingene er også en viktig faktor for å gi et realistisk bilde av ytelsen i trådløse nettverk.

3.3.2 Konfigurasjoner

I testoppsettet inngår hovedsaklig to iPAQ-er, og to arbeidsstasjoner. Disse maskinene kobles sammen i forskjellige konfigurasjoner — ulike nettverkstyper og klient-tjener-sammenkoblinger — for å gjøre målinger med

Maskin	*CPU*	*Minne*	*Støttet nettverk*
ipaq/ipaq2	SA-1110 206MHz	64MB	Wi-Fi, Bluetooth, IrDA
pc1	Pentium II 350MHz	512MB	Ethernet, Wi-Fi, Bluetooth
pc2	Pentium III 500MHz	384MB	Ethernet/Wi-Fi, IrDA
Logger	Pentium III 450MHz	512MB	Ethernet

Tabell 3.5: Maskiner brukt under målinger

	Maskinkonfigurasjon (klient - tjener)			
Nettverk	ipaq - ipaq2	ipaq - PC	PC - ipaq	PC - PC
lokal	1			2
Wi-Fi (Managed)	3	4	5	
Wi-Fi (Ad Hoc)	6			
Bluetooth (1-slot)	7	8	9	
Bluetooth (5-slot)	10	11	12	
IrDA (SIR/FIR?)	13	14	15	

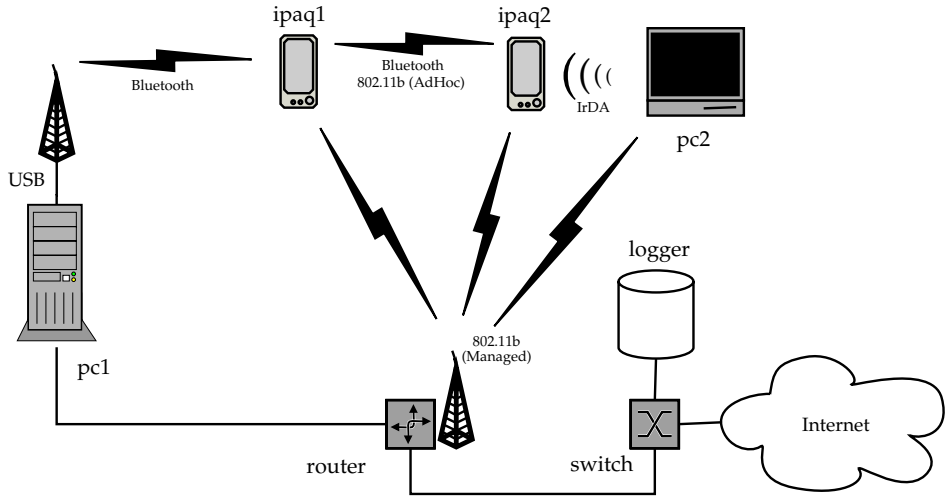
Tabell 3.6: Testkonfigurasjoner (nummerert for referanse). "PC" tilsvarer pc2 for IrDA, og pc1 for de øvrige.

CORBA-implementasjonene. I tillegg til disse fire maskinene bruker vi en arbeidsstasjon til logging av måledata. Tabell 3.5 viser maskinene som inngikk i testoppsettet.

Nettverkstypene som knytter disse maskinene sammen er Wi-Fi, Bluetooth og IrDA. Testoppsettet lot oss dermed gjøre ytelsesmålinger over disse tre nettverksteknologiene; iPAQ-ene (*ipaq* og *ipaq2*) kunne kommunisere sammen over Wi-Fi *Managed* og *AdHoc*, Bluetooth *Single slot* og *5 slot*, og IrDA *FIR* og *SIR*. Mot arbeidsstasjonen (*pc1*) ville iPAQ-ene kunne benytte seg av Wi-Fi *Managed*, samt Bluetooth *Single slot* og *5 slot*. Tabell 3.6 de ulike testkonfigurasjonene.

Arbeidsstasjonen med innebygget IrDA (*pc2*) ble satt opp til å bruke SIR til målinger mot iPAQ og Wi-Fi-forbindelsen til logging av måledata. *pc1* ble utstyrt med et Bluetooth USB-adapter, og koblet til Wi-Fi-ruteren via Ethernet for å kunne kommunisere med iPAQ-ene. Logge-maskinen ble koblet til ruteren via en 100 Mbit switch, som også var tilkoblet Internett. Figur 3.3 viser sammenkoblingen.

Ettersom iPAQ-ene har begrenset med lagringsplass, valgte vi å logge nett-



Figur 3.3: Testoppsett

verkstrafikk og systembelastning på en separat arbeidsstasjon. Loggen ble sendt over Wi-Fi-forbindelsen for iPAQ, og via Ethernet for arbeidsstasjonen — i begge tilfeller via Wi-Fi-ruteren.

3.3.3 Svakheter ved testoppsettet

Noen av våre designavgjørelser kan ha virket inn på ytelsesanalysen, og vi gir her en kort oversikt over aspekter ved testoppsettet vi mener kan ha gitt utslag på målingene.

- Skriptene, som på både klient- og tjener-siden logger CPU-, minne-, og nettverksforbruk, krever CPU-ressurser. For disse skriptene er det et mål i seg selv å være så enkle som mulig, slik at de ikke bruker unødvendig CPU-kraft.
- På iPAQ-ene logget vi måledata over Wi-Fi-forbindelsen. For målingene hvor ORB-ene kommuniserer over den samme forbindelsen, kunne dette virke inn på ytelsesmålingene. Vi har derfor gjort målinger av hvor mye data som blir logget, slik at dette kan medberegnes i analysen.
- Under Wi-Fi-testene er det en mulighet for at enheter som ikke inngår i testoppsettet, men som befinner seg seg i nærheten, kommuniserer på samme frekvens. For å unngå at målingene blir påvirket,

bør man sørge for at ingen andre Wi-Fi-enheter er i bruk. Vi valgte også å sperre Wi-Fi-ruteren for andre MAC-adresser enn de vi selv benyttet.

- For Bluetooth-testene skjer logging over Wi-Fi samtidig med bruk av Bluetooth. Disse nettverksteknologiene bruker samme frekvensbånd, og logging av måledata kan med andre ord påvirke Bluetooth-ytelsen.
- Under våre ytelsesanalyser var switchen, og dermed resten av testnettverket, hele tiden tilkoblet Internett. Maskiner involvert i målinger kan derfor ha hatt sporadisk nettverkstrafikk med maskiner utenfor testnettet. Maskinen vi logger måledata på er en arbeidstasjon i bruk til daglig, og vi har derfor ikke kunnet isolere nettverket helt. Vi anså imidlertid faren for at eventuell nettverkstrafikk skulle ha innflytelse på måledata som liten, da testmaskinene ikke selv kjørte, eller var avhengige av tjenester utenfra. Logging av nettverkstrafikken på både klient- og tjenersiden ville dessuten avsløre om én av maskinene hadde unormalt høy nettverkstrafikk under målingene. Tilsvarende ville sammenlikninger av total systembelasting kunne gi slike indikasjoner.
- Eventuelle *cron-jobber*[6] kunne ha innvirkning på måledata. Vi sørget imidlertid for at det disse jobbene ikke ble kjørt under målingene.

Med utgangspunkt i ressursene vi hadde til rådighet, mener vi at dette testoppsettet, med de forhåndsregler som er tatt, dannet en akseptabel plattform for våre målinger.

3.4 Sammendrag

I dette kapittelet har vi presentert hvilke krav som settes til testoppsettet for at vi skal nå vårt hovedmål. Kravene innbefatter maskin- og programvare og vi har redegjort for hvilke løsninger som har vært vurdert. Av maskinvare, har vi valgt iPAQ og Intel-baserte arbeidsstasjoner, med Wi-Fi- og Bluetooth-nettverk. For programvaren falt valgene på Orbix/E og MICO, og analyseverktøyet *Open CORBA Benchmarking*. Nærmere beskrivelse av programvaren er presentert i neste kapittel. Videre har vi skissert

[6]Jobber som kjøres regelmessig på UNIX-systemer

testoppsettet med de forskjellige kombinasjonene av program- og maskinvare, samt påpekt oppsettets svakheter.

Kapittel 4

Beskrivelse av verktøy

I dette kapittelet vil vi gi en grundig beskrivelse av verktøy vil vi har valgt til ytelsesmålingene. Vi har valgt å bruke måleverktøyet *Open CORBA Benchmarking*, og vil derfor gå nærmere inn på hvordan dette verktøyet måler forskjellige aspekter ved ytelsen til en CORBA-implementasjon. Vi har valgt Linux som testplattform, og vi vil her beskrive relevante deler av Linux-installasjonen i detalj.

4.1 Open CORBA Benchmarking

Open CORBA Benchmarking er et forskningsprosjekt av Petr Tuma og Adam Buble ved Charles University i Praha. I *Technical Report on Open CORBA Benchmarking*[30] beskriver de hvordan de har delt inn målgruppen for CORBA-analyseverktøy i to, ORB-brukere og ORB-utviklere, for å unngå at verktøyet blir for stort og uhåndterlig. De har foreløpig implementert bruker-versjonen, som består av flere isolerte, enkle tester.

Hovedkategoriene som måles er *systemytelse*, *metodekall*, *gjennomstrømning*, *skalerbarhet*, *parallellitet* og *kombinert*. Flere av disse er delt inn i delmålinger, hvor enkelte testparametre varieres. For hver av delmålingene kalles en metode, `MEAMeasure()`, som fungerer likt uavhengig av kategori, med unntak av responstid-målingen. For parallellitet- og kombinertmålinger benyttes en tilsvarende funksjon som støtter tråder.

`MEAMeasure()` starter hver måling med en *innkjøring* før den setter i gang med hovedmålingen. Denne innkjøringen har som mål å få systemet må-

Løkken som måler heltallsytelse

```
for (i = 0; i < 1000; i++) {
    iA += iB * iC;
    iB *= iC + iA;
    iC /= iA + iB;
};
```

Programeksempel 4.1: Utdrag fra heltallsmåling

lingene utføres på over i en *stabil tilstand*[34], og måleresultater fra denne fasen blir ikke brukt som datagrunnlag. Innkjøringsfasen varer i minimum 20 sykler, eller maksimalt 2 minutter. Deretter følger hovedfasen, hvor antallet sykler og maksimumstid er henholdsvis 100 og 10 minutter. For hver sykel utføres én operasjon, og tidsforbruket måles. Fem slike målinger blir utført for hver kategori. For hver av disse lagres minimum- og maksimumsverdier, i tillegg til at gjennomsnittet av alle 100 syklene blir beregnet. Disse tre verdiene blir lagret sammen med systemlasten før og etter målingen.

I det følgende beskrives de forskjellige hovedkategoriene med tilhørende delmålinger. Med unntak av for den første kategorien, *systemytelse*, omtales måling av CORBA-ytelse.

4.1.1 Systemytelse

Målingene av systemytelsen består av en rekke små, separate målinger av prosessor-, minne-, tråd- og socketytelse. Målingene blir først utført på klientsiden, og deretter på tjenersiden. Ingen av målingene gjør bruk av noe CORBA-funksjonalitet.

Prosessor

Analysen av prosessorytelse består av tre delmålinger kalt *Processor Move*, *Processor Integer* og *Processor Float*. *Processor Move* består i å allokere en minneblokk, hvorpå første halvdel av denne kopieres til siste halvdel ved kall på `memcopy()`. Deretter kopieres dataene tilbake. Dette gjøres først med minneblokker på 16KB, og deretter med 512KB.

Neste delmåling, *Processor Integer*, utfører aritmetiske operasjoner på heltall. Operasjonene bruker tre heltall, `iA`, `iB` og `iC` med utgangsverdiene

Utdrag fra minnetesten

```
for (i = 0 ; i < sArgs.iCount ; i ++)
  sArgs.apBlocks[i] = malloc(sArgs.iSizeMin +
                             (RNDGetRandom() & sArgs.iSizeMask));
for (i = 0 ; i < sArgs.iCount ; i ++)
  free (sArgs.apBlocks [i]);
```

Programeksempel 4.2: Utdrag fra minnetesten

123, 456 og 789. En løkke går 1000 ganger gjennom de samme operasjonene, men med ulike verdier for hver iterasjon, da variablene ikke nullstilles. Denne løkken vises i programeksempel 4.1.

I *Processor Float*-delmålingen utføres de samme operasjonene som i heltallstesten, men denne gang med flyttall. I tillegg er løkken av tidsmessige årsaker begrenset til 100 iterasjoner.

MeasureMemory

Måling av ytelsen til minnet baserer seg på allokering av mengder med minneblokker av økende størrelse. Det kjøres tre *runder*, hvor det i den første allokeres kun én minneblokk. I de etterfølgende blir det opprettet henholdsvis 100 og 10000. For hver av disse rundene utføres målinger med tre forskjellige *grunnstørrelser*. I den første brukes minimumsstørrelsen 32 bytes; deretter 512 og 4096. Denne størrelsen blir for hver allokering øket med en tilfeldig størrelse innenfor en viss ramme, gitt ved en *maske*, og resulterer i minneblokker på opptil 96, 1.536 og 12.288 bytes. Et utdrag fra koden vises i programeksempel 4.2.

MeasureThread

Operativsystemets evne til flertrådskjøring kan være avgjørende for hvor godt en ORB håndterer samtidige forspørsler fra forskjellige klienter. Målingene her berører to forskjellige aspekter ved tråder: *livssyklus* og *synkronisering*, hvor sistnevnte innbefatter trådbytte og trådlåsing.

Livssyklus-målingen utføres ved å opprette flere tråder og deretter avslutte dem. Dette utføres i to omganger; først med 8 tråder, så med 128. Synkroniseringsoperasjoner måles ved å repetere en *lock/unlock-operasjon*, hvor systemkallene `pthread_mutex_lock()` og

Fra	*Til*	*Økning*
0	5	1
10	50	10
100	500	100
1.000	5.000	1.000
10.000	50.000	10.000

Tabell 4.1: Størrelsen på parametrene (tall i bytes).

`pthread_mutex_unlock()` brukes, 100 ganger.

MeasureSocket

Socket-målingene foregår ved at pakker med tilfeldige data i tre forskjellige størrelser, 16B, 512B og 16KB, sendes til tjenersiden. På tjenersiden returneres dataene umiddelbart, og klienten registrerer hvor lang tid hele operasjonen har tatt. Til sending og mottak brukes metodene `send()` og `recv()`.

4.1.2 Metodekall (invocation)

Metodekall-målingen er den eneste av målingene som ikke ikke benytter seg av metoden `MEAMeasure()`. Den utfører 100 innkjøringssykler, for deretter å utføre 10.000 ping-operasjoner mot tjenersiden. Dette gjøres ved å kalle en metode hos tjeneren uten argumenter. Denne metoden returnerer uten å utføre noen operasjoner. På klientsiden registreres tidsforbruket for hele operasjonen.

4.1.3 Gjennomstrømning (marshalling)

Gjennomstrømning blir målt ved å sende pakker med parametrene av varierende størrelse, for å se hvordan økende datastørrelse påvirker responstiden. Først kjøres en test hvor klienter sender data som tjeneren dropper. Deretter kalles en metode uten argumenter på tjenersiden, som returnerer data. Begge disse målingene kjøres med økende størrelse på parametrene. Tabell 4.1 viser hvilke størrelser som brukes.

4.1.4 Skalerbarhet (dispatching)

Skalerbarhet måles ved at det opprettes flere og flere objekter på tjeneren, som det så måles responstid mot. For at denne målingen ikke skal ta for lang tid blir det i OCB gjort målinger mot et tilfeldig valgt objekt i stedet for mot alle. Dette gjøres fordi et kall mot hvert objekt når antallet objekter blir høyt vil føre til at målingene tar uforholdsmessig lang tid. Antallet objekter blir økt fra 1000 til 5.000 i intervaller på 1.000, og deretter fra 10.000 til 50.000 i intervaller på 10.000.

4.1.5 Parallellitet

For å måle hvordan tjeneren håndterer flere samtidige forspørsler fra flere klienter, blir et økende antall tråder opprettet på klientsiden. Dette er en forenkling, og simulerer flere klienter ved bruk av tråder, som hver utfører en enkel responstid-måling mot tjeneren. Tallet på antall tråder går fra 1 til 5, og deretter fra 10 til 50 i intervaller på 10, for til slutt å avslutte med 100.

4.1.6 Kombinert

I kombinertmålingene kombineres målemetodene fra *gjennomstrømning*, *skalerbarhet* og *parallellitet*. Først utføres en måling hvor gjennomstrømning og parallellitet kombineres. Her varierer antallet tråder mens størrelsen på argumentene endres. Deretter opprettes nye objekter på tjeneren for å kjøre målinger på skalerbarhet og parallellitet. Dette følges så opp ved å kjøre skalerbarhetmålinger med varierende argumentstørrelse, og til slutt testes gjennomstrømning, skalerbarhet og parallellitet samtidig.

4.2 Linux på iPAQ

Bruk av operativsystemet Linux på iPAQ er et forholdsvis nytt prosjekt, og det jobbes kontinuerlig med å tilpasse Linux-kjernen til nye modeller. Folk fra Compaqs *Cambridge Research Lab* vedlikeholder en tilpasset versjon av ARM-grenen av Linux 2.4. Denne inneholder en rekke kjerne-moduler —

Program(pakke)	*Versjon*
Linux-kjerne	2.4.18-rmk3-hh8
libc6	2.2.5-6
BlueZ utils	2.0-pre9
rfcommd	1.1

Tabell 4.2: Versjonsnummer for program(pakker) under Linux på iPAQ.

drivere for iPAQ-maskinvare. De samme menneskene er en del av drivkraften bak Linux-distribusjonen *Familiar*, som bygger på *Debian Linux*, men er tilpasset maskiner med begrenset plass.

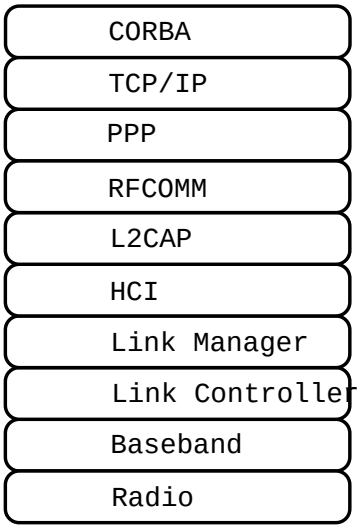
På våre iPAQ-er har vi installert en distribusjon basert på versjon 0.5.2 av *Familiar Linux*. Etter installasjon har de fleste av programpakkene blir oppgradert til nyere versjoner gjennom Familiars pakkesystem (tabell 4.2 viser versjonsnummer for sentrale pakker). Kjernen vi har brukt er en tilpasset versjon av 2.4.18-rmk3. Denne støtter alt av innebygde maskinvarefasiliteter for iPAQ H3870, blant annet Bluetooth. I tillegg støttes en god del ekstraustyr; for eksempel *PC Card sleeve*, som vi har benyttet for å få brukt nettverkskort.

Linux 2.4-kjerner har innebygget støtte for PC Card-nettverkskortet vi benytter, Cisco Aironet 350, gjennom modulene `airo_cs` og `airo`. Ved hjelp av verktøyet `iwconfig` kan en konfigurere kortet. For vårt vedkommende var muligheten til å slå av og på kryptering, samt det å sette operasjonsmodus til *Ad-Hoc* eller *Managed* det vi brukte `iwconfig` til.

BlueZ, den offisielle Linux Bluetooth protokollstakken, er også inkludert i Linux 2.4-kjernene. For å konfigurere Bluetooth-maskinvaren trenger man i tillegg til kjerne-modulene verktøy fra *BlueZ utils*-pakken. I tillegg behøver man demonen `rfcommd` for å få emulert et serielt grensesnitt, som *PPP* kan brukes over. Figur 4.1 viser hvor *RFCOMM*- og *PPP*-laget befinner seg i protokollstakken.

På iPAQ H3870 er Bluetooth-maskinvaren (ALPS UGTZ4) forbundet med resten av iPAQ-en via et serielt grensesnitt. Brikken (CSR), som tar seg av dette, måtte omprogrammeres for å fungere med Linux, og ble ved bruk av verktøyet `pskey` satt til *H4*-modus. Denne kan også settes til å operere på 230.400 bps, mot 115.200 som forhåndsprogramert fra fabrikken.

For å kunne bestemme om *single-slot*- eller *5-slot*-pakker skal benyttes under målinger benytter vi verktøyet `hciconfig`. Et eksempel på hvordan



Figur 4.1: Bluetooth protokollstakk, med overliggende PPP- TCP/IP- og CORBA-lag.

Eksempel på setting av pakketype

```
$ hciconfig hci0 ptype DH1,DM1
```

Programeksempel 4.3: Setting av Bluetooth-pakketype i Linux

dette gjøres vises i programeksempel 4.3.

IrDA-brikkesettet som finnes i iPAQ H3870 støtter både *Serial IR* (SIR) og *Fast IR* (FIR).

Kapittel 5

Implementasjon

Bruk av av ORB-ene, maskinvaren og måleverktøyet i kombinasjon krevde enkelte tilpasninger for å fungere sammen. Måleverktøyet var ikke konfigurert for å brukes på ressurssvake enheter, og Orbix/E var heller ikke konfigurert for å støtte analyseverktøyets behov for samtidige objekter. *Open CORBA Benchmarking* består av to deler: én del for ytelsesmålinger og én del for analyse av dataene. Analysebiten er implementert som en webtjeneste, men denne ga oss ikke god nok grafisk presentasjon. Vi valgte derfor å implementere analysedelen selv slik at vi fikk full kontroll over genereringen av grafer og statistikk. Analyseverktøyet gir oss ingen data om forbruk av systemressurser under ytelsesmålingene, og vi har derfor valgt å skrive egen programvare for logging og analysering av systemressursforbruket. Dette kapittelet gir en beskrivelse av tilpasninger vi har gjort, samt en beskrivelse av all programvaren vi selv har skrevet. Vi peker også på svakheter ved metodene vi har valgt.

5.1 Tilpassing av programvare

Open CORBA Benchmarking er skrevet for arbeidsstasjoner som er langt mer ressurssterke enn iPAQ, og det har derfor vært nødvendig med enkelte tilpasninger for å få analyseverktøyet til å fungere på iPAQ. Hovedsakelig gjelder dette enkelte grenseverdier for forskjellige ytelsestester, som måtte justeres for å ikke overbelaste iPAQs minneressurser. I tillegg var det nødvendig å tilpasse OCBs kode for måling av tidsforbruk, ettersom OCB bruker maskinvare-spesifikke instruksjoner.

5.1.1 Open Corba Benchmarking

iPAQ H3870 er utstyrt med 64MB RAM, som under Linux blir delt dynamisk mellom allokerbart minne og en *ramdisk*[1] som i størrelse er oppad begrenset til 32MB. Ved å fjerne alle unødvendige data fra ramdisken, samt begrense kjørende prosesser til de aller nødvendigste, endte vi opp med omtrent 50MB ledig minne tilgjengelig til kjøring av Open Corba Benchmarking. Slik OCB opprinnelig var konfigurert var kravet til minne høyere enn det som var tilgjengelig, Resultatet at dette var at OCB avbrøt ytelsesmålingene når iPAQ-en gikk tom for minne. Under innledende testing skjedde dette ved flere tilfeller.

Første stans skjedde under minne-testen. Den tredje og siste delen av denne testen, som fyller minnet med store blokker med data, brukte opp alt minnet slik at prosessen avsluttet. I OCB-kildefilen `shr_defs.cc` defineres verdien `SYS_MEM_LargeBlockNum`, som bestemmer hvor mange blokker av 4KB størrelse som skal opprettes. Verdien var opprinnelig satt til 10.000, men ved å redusere denne til 9.000 fullførte testen.

Under skalerings- og kombinert-testen gikk også iPAQ-en tom for minne. I filen `all_defs.h` defineres antall objekter som skal opprettes under disse testene. Konstanten `SAM_ScaleMaxSize` var opprinnelig definert til 50.000, men vi justerte den til 20.000. MICO hadde ingen problemer med 20.000 objekter, og vi kjørte også noen testmålinger med bruk av 30.000 objekter. Orbix/E klarte derimot ikke å opprette flere enn 10.000 objekter uten at målingen feilet.

Open Corba Benchmarking bruker maskinvarespesifikke instruksjoner for måling av tidsforbruk for å få resultatet så nøyaktig som mulig. OCB distribueres med støtte for *Intel i386* og *Sun SPARC*. Intel-delen bruker instruksjonen `RDTSC` (read time-stamp counter), som ble introdusert i Pentium-familien. Denne instruksjonen finnes ikke for StrongARM-arkitekturen, og OCB lot seg av den grunn ikke kompilere. Vi ble derfor nødt til å benytte oss av `gettimeofday()`-systemkallet for kjøring på iPAQ. Denne metoden har dessverre noe mindre presisjon enn RDTSC-metoden.

Den første delen av *Open Corba Benchmarking*-målingene består av systemtester. Her måles ytelsen til prosessor, minne, flertrådskjøring og sockets. Bortsett fra socket-testen, som er avhengig av hvilken nettverksteknologi

[1] *Ramdisk* er en emulering av fysisk lagringsplass (disk) i RAM.

som benyttes, gir maskinvaretestene samme resultater hver gang de kjøres på samme maskin. Ettersom maskinvaretestene er relativt tidkrevende og vi skulle kjøre testene mange ganger, modifiserte vi OCB slik at det var mulig å hoppe over disse testene. Socket-testen er som nevnt avhengig av valg av nettverk og vi valgte å la denne testen bli utført for hver måling.

Open CORBA Benchmarking gir ingen informasjon om når den starter de forskjellige testene, så vi valgte å legge inn logging av dette. Før hver av hovedmålingene logges navnet på målingen, samt antall sekunder siden *Unix epoch*[2] til *standard out*.

5.1.2 Orbix/E

Innledende OCB-testkjøringer med Orbix/E viste at ORB-en fikk problemer ved skaleringstesten. Under denne testen opprettes en mengde objekter, og etter nærmere undersøkelse fant vi at det ikke ble opprettet mer en 1.000 objekter før Orbix/E fikk problemer. Dokumentasjonen forteller svært lite om begrensninger i antall samtidige objekter, og vi ble derfor nødt til å studere kildekoden i detalj. I filen `obec/include/OBEC/Defs.h` fant vi en konstant for dette, `MAX_ACTIVE_OBJECTS`, som var satt til 1.000. Vi endret verdien av denne til 50.000, som er det maksimale antallet objekter OCB ønsker å opprette og dette løste problemet.

5.2 Egne måleverktøy

Som nevnt i innledningen til dette kapittelet har ikke *Open CORBA Benchmarking* noen rutiner for å logge forbruk av systemressurser under målinger. Vi har behov for data som potensielt kan hjelpe til med å forklare resultater fra ytelsesmålingene.

Proc-filsystemet i Linux gir tilgang til informasjon direkte fra operativsystemkjernen. Blant annet finnes det detaljerte data for hver enkelt prosess, samt de forskjellige subsystemene i kjernen. Jevnlig logging av disse dataene gir oss mulighet til å kartlegge prosessenes ressursforbruk.

Vi har valgt å lage to separate skript, ett for logging av prosessinformasjon

[2]Unix epoch = 1. januar 1970 kl. 00:00:00

og ett for nettverksdata. Skriptene går i en løkke og logger tidspunktet etterfulgt av hele innholdet av de respektive filene, med ett sekunds mellomrom. Dataene sendes til *standard out*. Skriptene er gjengitt i sin helhet i tillegget (delkapittel B.2.1 og B.2.2).

5.2.1 Logging av prosessor- og minneforbruk

Hver prosess har i Linux en egen katalog i det virtuelle filsystemet `/proc/`. I denne katalogen ligger filen `stat`, som inneholder én linje med statusinformasjon om den respektive prosessen. Blant annet inneholder filen data om hvor mye minne prosessen bruker, hvor mye tid den har blitt tildelt av operativsystemet i tillegg til en del annen informasjon. Eksempel på innhold av `stat`-filen finnes i programeksempel 5.1 (videre forklaring av innholdet kommer i delkapittel 5.3 som omhandler behandling av innsamlede data).

Informasjon om maskinens *last* finnes i filen `/proc/loadavg`. *Last* angis som et desimaltall, og angir hvor mange prosesser som ligger i operativsystemets kjørekø. Under normale omstendigheter[3] vil en last på 1,0 bety at prosessoren er fullt belastet. Siden lasten kan variere kraftig over et kortere tidsrom viser `/proc/loadavg` gjennomsnittlig last for siste 1, 5 og 15 minutter (eksempel på innholdet av filen vises i programeksempel 5.1).

Skriptet `sysusage.sh` (også vist i programeksempel 5.1) skriver ut prosess- og last-informasjonen til *standard out*, og startes ved å angi prosess-id for ønsket prosess, samt antall sekunder for forsinkelsen mellom hver logging. Sistnevnte parameter er valgfri, standardforsinkelsen er satt til ett sekund. For å få komplett informasjon om CPU-forbruket til alle prosessene som kjøres under ytelsesmålingen, må `sysusage.sh` logge `stat`-filen til den første prosessen som startes, foreldreprosessen. Denne inneholder CPU-forbruket til alle sine barn i tillegg til eget forbruk.

Da vi studerte `status`-filene til OCB-prosessene, som angir samme informasjon som `stat`-filen, men på et mer lettlest format, fant vi at at alle prosessene brukte eksakt like mye minne. Av det vi har funnet av dokumentasjon i `man`-sidene til `proc`, samt i Linux-kildekoden[4], skal det rapporterte minneforbruket ikke dreie seg om delt minne. I Linux, som andre unix-operativsystemer, opprettes nye prosesser ved hjelp av *fork*-

[3] En prosess som venter på I/O vil trekke opp lasten uten at den belaster prosessoren.
[4] linux/fs/proc/array.c

Innhold av /proc/{PID}/stat

```
$ cat /proc/7445/stat
7445 (emacs21) S 1 7445 7349 0 -1 0 7662 70640 2426 108808 607435 64058 2982 447 9 0 0 10 25378438 18604032 \
3669 4294967295 134512640 135912432 3221222064 3221212560 1108220302 0 0 0 1367441149 3222560584 0 0 17 1
```

Innhold av /proc/loadavg

```
$ cat /proc/loadavg
0.12 0.11 0.09 3/143 20344
```

Prosess (sysusage.sh B.2.1)

```
26 until [ ! -e "/proc/$1" ]; do
27     echo ‘date +%s‘" "‘cat /proc/$1/stat‘" "‘cat /proc/loadavg‘
28     sleep $DELAY
29 done
```

Start av sysusage.sh

```
# Skriver ut /proc/7445/ til stdout hver 3. sekund.
$ sysusage.sh 7445 3
```

Programeksempel 5.1: Datafiler, skript og kommandolinjeeksempel for prosessinformasjon.

kallet. Dette kallet lager en barneprosess som er en identisk kopi av foreldreprosessen[35]. Om barneprosessen ikke endrer minneforbruket, vil den da bruke like mye minne som foreldreprosessen. Til tross for dette antar vi at minnet, som proc-filene rapporterer at prosessen bruker, er delt mellom dem, ettersom summen av minneforbruket til alle maskinens prosesser da stemmer overens med det totale minneforbruket på maskinen.

5.2.2 Logging av båndbreddeforbruk

Nettverksgrensesnittene er ikke knyttet til spesifikke prosesser, og har derfor en egen katalog i proc-filsystemet `/proc/net/`, hvor filen `dev` er plassert. Denne filen inneholder informasjon om hvert av operativsystemets nettverksgrensesnitt. Filen inneholder en forklaring av filformatet, etterfulgt av data for hvert grensesnitt — én linje per grensesnitt (se programeksempel 5.5). Vi er hovedsakelig interessert i elementene som inneholder informasjon om hvor mye data som er sendt og mottatt for hvert grensesnitt, slik at vi kan kalkulere båndbreddeforbruket.

Skriptet for logging av nettverksgrensesnittene fungerer på samme måten som logging av prosessinformasjonen (se programeksempel 5.1 og 5.5). Forskjellen ligger i at skriptet ikke avslutter automatisk fordi `/proc/net/dev` ikke er tilknyttet noen spesifikk prosess.

```
$ cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls
    lo:227318538  595506   0    0    0     0          0         0 227318538 595506    0    0    0     0 0 0
  eth0:2068618522 25331053 0    0    0     0          0         0 2883162652 42668682 0    0    0     0 0 0
```

Nettverk (netusage.sh B.2.2)

```
14 while [ 1 ]; do
15     echo "Time: "‘date +%s‘
16     cat /proc/net/dev
17     sleep $DELAY
18 done
```

Eksempel på start av netusage.sh

```
# Skriver ut /proc/net/dev til stdout hver 3. sekund.
$ netusage.sh 3
```

Programeksempel 5.2: Datafiler, skript og kommandolinjeeksempel for nettverksgrensesnittene. **Merk:** De to siste kolonnenavnene fra datafilen, *carrier* og *compressed*, fikk ikke plass på utskriften.

5.2.3 Svakheter

Ettersom iPAQ har begrenset med lagringsplass, valgte vi å logge til en NFS-montert disk via Wi-Fi-grensesnittet. Ved bruk av Bluetooth, vil loggingen ikke bruke av båndbredden som benyttes til ytelsesanalysen, men når analysen kjøres via Wi-Fi vil loggingen legge beslag på noe av båndbredden. Forbruket av båndbredde er avhengig av hvor mye som logges. `/proc/{PID}/stat` inneholder rundt 100-200 byte, mens størrelsen til `/proc/net/dev` varierer fra maskin til maskin etter hvor mange nettverksgrensesnitt som er konfigurert. Tabell 5.1 viser størrelsen på `/proc/net/dev` for maskinene vi kjørte ytelsestestene på.

Logger vi nettverksinformasjonen hvert sekund blir båndbreddeforbruket fra iPAQ-en rundt regnet 8 kbps i tillegg til TCP/IP-overhead. En løsning for å redusere datamengden er å fjerne konfigurasjonen for de nettverksgrensesnittene som ikke er i bruk under målingene.

Vi har ingen mulighet til å knytte bruk av båndbredde opp mot prosesser, ettersom `/proc/net/dev` kun rapporterer hvor mye informasjon som totalt er sendt og mottatt over de forskjellige nettverksgrensesnittene. Med andre ord kan andre prosesser på maskinen innvirke på båndbreddeforbruket uten at dette fremgår av loggene. For å minimere dette problemet, har vi stanset alle unødvendige prosesser på maskinene.

Maskin	*Størrelse (bytes)*
pc1	457
pc2	455
ipaq	944
ipaq2	942

Tabell 5.1: Størrelsen til `/proc/net/dev`

De ovennevnte skriptene ble implementert så enkelt som mulig for å holde ressursforbruket på et lavt nivå. Shell-skriptene dumper innholdet av filer til *standard out* og shellet redirigerer dette videre til fil. Dette er en veldig enkel metode å implementere, og virker som en god løsning ved første øyekast. Ulempen er at hver gang skriptet skal logge innholdet av en *proc*-fil, opprettes det nye prosesser. `sysusage.sh` oppretter tre nye prosesser (`date` og to ganger `cat`), og `netusage.sh` oppretter to. Det er forbundet en del *overhead* med opprettingen av nye prosesser, og dette slår ekstra ut på treg maskinvare. På arbeidsstasjonene var CPU-forbruket fra 0,02 til 0,03 sekunder per løkke (se tabell 5.2). Kjøres løkken hvert sekund, vil dette bety et prosessorforbruk på 2-3%.

Dessverre gjorde vi ikke målinger av tidsforbruket på iPAQ før vi hadde gjennomført OCB-målingene. I ettertid fant vi at skriptene brukte mellom 0,09 og 0,10 sekunder, noe som tilsvarer 9-10% forbruk av prosessoren når loggingen foretas hvert sekund. En mer effektiv løsning er å implementere et program som foretar loggingen uten opprettelsen av nye prosesser. For å få et sammenlikningsgrunnlag, implementerte vi et testprogram i C for å måle tidsforbruket. Tidsforbruket for dette programmet lå på 50-30% av forbruket til shell-skriptene, og er derfor et langt bedre alternativ.

`netusage.sh` er litt mer effektivt enn `sysusage.sh` ettersom den oppretter to nye prosesser, mot tre, for hver logging. Selv om dataene som logges er mer omfattende for nettverksinformasjonen viser tabell 5.2 at `netusage.sh` er raskest, med 1-2% forbruk på arbeidsstasjon mot 7-8% på iPAQ.

For ytteligere å redusere forbruket av prosessorkraft, kunne all loggingen vært gjort av ett skript i stedet for to — gjerne implementert i C. Etter å ha gjennomført CORBA-målingene, valgte vi å studere påvirkningen av loggeskriptene nærmere og skrev derfor en C-implementasjon av loggeskriptene (se programeksempel 5.2.3), for å ha som sammenlikningsgrunnlag. Måletallene i tabell 5.2 viser at det CPU-forbruket er markant lavere ved bruk av C-programmet, noe som betyr at vi nok heller burde benyttet den-

	Maskin	*sysusage.sh*	*netusage.sh*
Shell	pc1	0,02-0,03s	0,01-0,02s
	pc2	0,02-0,03s	0,01-0,02s
	ipaq	0,09-0,10s	0,07-0,08s
C	pc1	0,00-0,01s	0,00-0,01s
	pc2	0,00-0,01s	0,00-0,01s
	ipaq	0,02-0,03s	0,03-0,04s

Tabell 5.2: Systemforbruket for sysusage/netusage, målt med `time`.

ne metoden ved logging av ressursforbruk.

Som nevnt venter loggeskriptene en fast tidsenhet mellom hver loggføring. Til dette benyttes systemkallet *sleep*, som resulterer at prosessen stanses av operativsystemet i et gitt antall sekunder. Kalles sleep med "1" som parameter, så stanses prosessen i *minimum* ett sekund — er systemet hardt belastet vil tiden mest sannsynlig bli noe lengre. Dette i kombinasjon med at vi ikke logger tidspunktet mer nøyaktig enn til nærmeste sekund, medfører en unøyaktighet i målingene. Resultatet av denne unøyaktigheten ser vi ved utregning av CPU-forbruket; verdien overstiger i enkelte tilfeller 100%.

5.3 Behandling av innsamlede data

Open CORBA Benchmarking har en egen resultatbase med tilhørende webgrensesnitt. Systemet baserer seg på innsending av den XML-baserte datafilen, som så lagres i en database. Etter kort tid får brukeren tilbake en URL til en presentasjon av den automatiserte analysen. Analysen består av grafer, nøkkeltall og tabeller. Til vårt formål er de genererte grafene av for dårlig kvalitet, og vi valgte derfor å utvikle egne verktøy for å erstatte den webbaserte analysetjenesten.

Kjøring av loggeskriptene omtalt i forrige delkapittel produserer fire datafiler: prosess- og nettverksdata for både klient og tjener. I tillegg genererer *Open CORBA Benchmarking* en XML-datafil med resultater fra ytelsesmålingene, samt to datafiler med utskrift av tidspunkter for de forskjellige testene. Vi har valgt å bruke programmet *Gnuplot* til visualisering av måledata, og innholdet av disse filene må derfor konverteres til datafiler som egner seg for generering av grafer med dette programmet.

ex_usage.c

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main(int argc, char *argv[]) {
5   int i = 0;
6   void cat_file(char *);
7
8   printf("%i\n", time(NULL));
9   for (i = 1; i < argc; i++) {
10    cat_file(argv[i]);
11  }
12 }
13
14 void cat_file(char *f_name) {
15  int c;
16  FILE *fp;
17
18  fp = fopen(f_name, "r");
19
20   while ((c = getc(fp)) != EOF) {
21     putc(c, stdout);
22   }
23
24   fclose(fp);
25 }
```

Programeksempel 5.3: Testprogram (C) for sammenlikning av tidsforbruk.

For at det skal være lettere å se endringer i forbruk av systemressurser i sammenheng med hvilken test som har blitt utført, har vi ved generering av *Gnuplot*-filer modifisert tidspunktene. Ved å la starttidspunktet for hver OCB-måling være nullpunktet for de tilhørende systemressurs-logger, har vi beregnet tiden for alle loggede data relativt til dette.

Utdrag fra en prosesslogg

```
1025605814 23093 (Client) D 23091 23091 22976 34820 22976 0 16 0 3 0 0 0 0 0 14 0 0 0 49492436 196608 2 \
4294967295 134512640 134600136 3221223952 3221223052 1108036714 0 0 6 0 3222425132 0 0 17 0 0.00 0.00 0.00 \
1/65 23100
```

Eksempel på kjøring med tidssynkronisering

```
$ gen_system_plots.pl --data s  --time 1028123013 \
  sysusage_client.data
```

Programeksempel 5.4: Data fra prosesslogg og konvertering av data, med tidssynkronisering.

Data	*Element nr.*	*Benevning*
Tidspunkt	1	Sekunder siden epoch
Brukertid	15	Tidsluker
Systemtid	16	Tidsluker
Barnas brukertid	17	Tidsluker
Barnas systemtid	18	Tidsluker
Minneforbruk (virtuell)	24	Bytes
Load (siste minutt)	41	Prosesser i kjørekø

Tabell 5.3: Aktuelle data fra prosessloggen.

5.3.1 Prosessor- og minneforbruk

Ut fra `/proc/{PID}/stat` kan vi lese hvor mange tidsluker en prosess har blitt tildelt av operativsystemet. Som tabell 5.3 viser, er det fire elementer som omhandler tildelte tidsluker: *brukertid* (utime), *systemtid* (stime), *barns brukertid* (cutime) og *barns systemtid* (cstime). Summen av alle disse elementene gir det totale antallet tidsluker prosessen og barneprosessene har brukt i både *kernel-* og *user space*. Siden loggen er generert basert på foreldreprosessen brukt i ytelsesmålingen, vil vi få en oversikt over hvor mye CPU-tid hele OCB-målingen har brukt. Linux bruker tidsluker med 1/100 sekunds lengde, og en prosess som har bruk 50 tidsluker i løpet av ett sekund har da lagt beslag på 50% av tilgjengelig prosessorkapasitet.

I tillegg til CPU-forbruk, inneholder `/proc/{PID}/stat` også informasjon om hvor mye minne som er allokert av prosessen. Som tabell 5.3 viser, inneholder element nr 24 antall bytes prosessen bruker av operativsystemets virtuelle minne.

For å generere plot-filer til *Gnuplot*, har vi laget et skript som genererer

Data	*Element nr.*	*Benevning*
Grensesnitt	1	Navn (eth0, ppp0)
Mottatt	2	Bytes
Sendt	10	Bytes

Tabell 5.4: Aktuelle data fra nettloggen.

to separate plot-filer: én plotfil for CPU-forbruk og én for minneforbruk. Skriptet, `gen_system_plots.pl` (gjengitt i tillegg B.3.1), kjøres separat for loggfilen til klienten og tjeneren.

5.3.2 Nettverk

Som for `stat`-filen er vi kun interessert i enkelte elementer fra `/proc/net/dev`. Tabell 5.3 gir en oversikt over hvilke dataelementer vi benytter oss av. Ettersom vi er interessert i å se det totale båndbreddeforbruket under målingene, beregner vi summen av datamengden som er sendt og mottatt via nettverksgrensesnittet. For Bluetooth-målingene benyttes `ppp0`-grensesnittet, mens Wi-Fi bruker `eth0`. Båndbreddeforbruket beregnes ved hjelp av likning 5.1.

$$bandbredde(kbps) = \frac{\frac{(sendt+mottatt)\times 8}{tid(s)}}{1024} \tag{5.1}$$

5.3.3 Data fra ytelsesmåling

Analysedelen av *Open CORBA Benchmarking* (se delkapittel 4.1) baserer seg på en database med et webgrensesnitt implementert med PHP. Vi forhørte oss med OCB-utviklerene om muligheten til å få tilgang til kildekoden til disse analyseverktøyene, men de ønsket ikke å gi ut dette. Som tidligere beskrevet gir ikke det webbaserte analyseverktøyet grafer som egner seg til vårt formål og vi valgte derfor å implementere tilsvarende skript i *Perl*. Hovedsaklig er vi interessert i å studere grafer med måledataene, og skriptene ble derfor skrevet for å generere filer som *Gnuplot* kan visualisere. Skriptet er gjengitt i sin helhet i tillegg B.3.2. Dokumentasjonen for måledataene finnes i tillegg C.

Eksempellogg

```
Time: 1025605816
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls
    lo:4279774368 79200356 0    0    0     0          0         0 4279774368 79200356 0    0    0     0 0 0
  eth0:1334052930 5791041  0    0    0     0          0         0 832213296 4601665   0    0    0     0 0 0
  ppp0:936823395 1049551  16    0    0     0          0         0 824837941  986390   0    0    0     0 0 0
```

Eksempel på kjøring med tidssynkronisering

```
# Starter generering av plot-fil, med synkroniseringstidspunkt.
# Resultatet skrives til standard out.
$ gen_system_plots.pl --data n  --time 1028123013 \
  netusage_server.data
```

Eksempel på plot-fil for båndbredde (kbps)

```
# Time  ppp0(t) ppp0(r)
31      5.09375 4.078125
32      8.5     8.5
33      6.78125 6.375
```

Programeksempel 5.5: Data fra logg, kjøring av konverteringsskript samt resultatdata. **Merk:** De to siste kolonnenavnene fra datafilen, *carrier* og *compressed*, fikk ikke plass på utskriften.

Open CORBA Benchmarking angir alle tidsmålingene i antall klokkesykler. Vi har valg å regne om responstidene til mikrosekunder, ettersom det er tidsenheten OCB bruker i de webbaserte analysene sine. Datafilen for målingene (XML) inneholder en *timer-tag* som angir hvor mange klokkesykler maskinen utfører per sekund (skala), som benyttes ved konvertering til mikrosekunder (likning 5.2). Eksempel på innhold i datafiler og omregning til finnes i programeksempel 5.6.

$$tid(\mu s) = \frac{klokkesykler}{skala} \times 10^6 \tag{5.2}$$

Metodekall

Invocation-testen viser hvor lang tid det tar ORB-en å svare ved et enkelt metodekall. I XML-filen er disse målingene pakket inn som *CDATA* i en *measurement*-tag (se tillegg C). Som nevnt i delkapittel 4.1.2 (side 52) er dette den eneste målingen vi har alle måledataene fra. Tilsammen blir det foretatt 10.000 målinger av responstiden. Fra måledataene beregner vi gjennomsnittsverdien, medianen og kvartilene. I tillegg visualiserer vi distribusjonen av målingene som en plottet graf. Programeksempel 5.7 viser

Eksempel på Timer-tag i måledataene.

```
<Timer Scale="500893162" Granularity="1"></Timer>
```

Omregning til mikrosekunder (B.3.2)

```
15 my $scale =
16     $doc->getElementsByTagName("Timer")->item(0)->
17     getAttributeNode("Scale")->getValue;

63 sub microseconds {
64     my ($tics) = @_;
65
66     return 0 if (!$tics);
67
68     # $tics devided by $scale, mulitiplied by 1.000.000.
69     return sprintf "%.0f", ($tics/$scale)*1000*1000;
70
71 }
```

Programeksempel 5.6: Data og skript for omregning fra klokkesykler til mikrosekunder.

et eksempel fra datafilen og rutinene for behandling av dataene. Figur 5.1 viser et eksempel på en grafisk fremstilling av distribusjonen av målingene.

Øvrige CORBA-målinger

Som nevnt har vi kun tilgang på alle måledata for *invocation*-testen. For *gjennomstrømning*, *dispatcher* og *parallellitet*, behandles måledataene av *Open CORBA Benchmarking* før de lagres i resultatfilen. Minimum-, maksimum- og snittverdier beregnes, og lagres gruppert som et XML-element, *Sample* (se programeksempel 5.8). Som nevnt innledningsvis i 4.1 (side 49) blir fem slike *Sample*-elementer lagret for hver delmåling (*Measurement*-elementet). Flere delmålinger blir gruppert i et *Benchmark*-element.

Minimumsverdien er den laveste verdien for alle *Sample*-elementene, mens maksimumsverdien er den høyeste verdien. Gjennomsnittsverdien beregnes ved å ta gjennomsnittet av alle *Sample*-elementenes lagrede gjennomsnittsverdi (den midterste verdien i *Sample*-elementet er gjennomsnittet). Programeksempel 5.8 viser rutinen for omregning av resultatdataene samt eksempel på en plot-fil som er generert ved hjelp av konverteringsskriptet.

Eksempel på resultat av responstidsmåling

```
<Benchmark Type="Ping Roundtrip">
  <Measurement>
    45244 52116 45278 45185 45177 45151 45207 45107 45192 45152
  </Measurement>
</Benchmark>
```

Beregning av gjennomsnitt median og kvartiler

```
292     @data = sort {$a <=> $b} @data;
293
294     # Median and quartiles
295     if ($length %2 == 1) {
296         my $med_pos = ($length/2)+0.5;
297         my $q1_pos  = $med_pos/2;
298         my $q3_pos  = $q1_pos + $med_pos-1;
299
300         #-1 since the first element of the array is numbered "0".
301         $median = $data[$med_pos-1]."\n";
302         $q1 = sprintf "%.0f",($data[$q1_pos-1]+$data[$q1_pos])/2;
303         $q3 = sprintf "%.0f",($data[$q3_pos-1]+$data[$q3_pos])/2;
304     }
305
306     else {
307         my $med_pos = $length/2;
308         my $q1_pos = ($med_pos/2)+0.5;
309         my $q3_pos = $q1_pos + $med_pos;
310
311         #-1 since the first element of the array is numbered "0".
312         $median = sprintf "%.0f",($data[$med_pos-1]+
313                                   $data[$med_pos])/2;
314         $q1 = $data[$q1_pos-1];
315         $q3 = $data[$q3_pos-1];
316     }
317
318     # Sum of all elements (for computation of average)
319     for my $elem (@data) {
320         $sum += $elem;
321     }
322
323     open(FILE, ">benchmark_stats.txt");
324     print FILE "Invocation\n";
325     print FILE "**********\n";
326     print FILE "Median: $median\n";
327     print FILE "Q1: $q1\n";
328     print FILE "Q3: $q3\n";
329     printf FILE "Average:  %.0f\n", ($sum/$length);
330     print FILE "Min: $data[0]\n";
331     print FILE "Max: $data[$#data]\n";
```

Programeksempel 5.7: Måledata fra *invocation* og rutine for beregning av snittverdi, median og kvartiler.
Merk: *Measurement*-elementet inneholder 10.000 målinger, mot 10 i dette eksempelet.

Eksempel på resultat av måling

```
<Benchmark Type="Sequence In">
<Measurement Size="0">
<Sample LoadBefore="19" LoadAfter="14">45913 54033 3486542</Sample>
<Sample LoadBefore="8"  LoadAfter="10">45594 53845 3555176</Sample>
<Sample LoadBefore="12" LoadAfter="13">43767 53622 3646901</Sample>
<Sample LoadBefore="10" LoadAfter="11">45620 53132 4144817</Sample>
<Sample LoadBefore="13" LoadAfter="7" >45613 53758 2487687</Sample>
</Measurement>
</Benchmark>
```

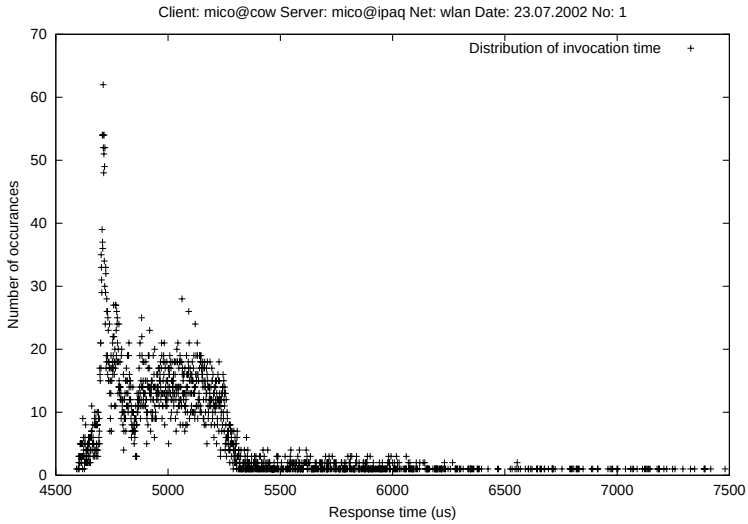
Beregning av min maks og gjennomsnitt av målinger

```
206     for (my $i = 0; $i < $samples->getLength; $i++) {
207         my $sample = $samples->item($i);
208         my $values = $sample->getFirstChild->getData;
209 
210         $values =~ m/([0-9]+)\s([0-9]+)\s([0-9]+)/;
211 
212         # Min values
213         if (!$data{'min'} || $1 < $data{'min'}) {
214             $data{'min'} = $1;
215         }
216 
217         # Max values
218         if (!$data{'max'} || $3 > $data{'max'}) {
219             $data{'max'} = $3;
220         }
221 
222         # Add all values
223         $data{'avrg'} += $2;
224 
225         $data{'load_before'} +=
226             $sample->getAttributeNode("LoadBefore")->getValue;
227         $data{'load_after'} +=
228             $sample->getAttributeNode("LoadAfter")->getValue;
229     }
230 
231 
232     $data{'min'} = &microseconds($data{'min'});
233     $data{'avrg'} = &microseconds($data{'avrg'}/$samples->getLength);
234     $data{'max'} = &microseconds($data{'max'});
```

Eksempel på genererte plot (gjennomsnittsverdier)

```
1 # Sequence In - avrg-values
2 # Size        Time (us)
3 0         107
4 1         108
5 2         108
```

Programeksempel 5.8: Data fra målinger, konverteringsskript (min, maks, snitt) og resultatdata.



Figur 5.1: Eksempel på plot av responstid, generert av *Gnuplot*.

Maskinytelse

Resultatene fra maskinvaretestene (*microbenchmarks*) lagres på samme format som de andre OCB-deltestene, og vi benytter derfor samme konverteringsrutiner for dataene. I stedet for å generere plot-filer for *Gnuplot*, genererer vi imidlertid tekstfiler med de behandlede resultatene. Programeksempel 5.3.3 viser et utdrag av denne resultat-filen.

Generering av plot

Som tidligere nevnt har vi benyttet programmet *Gnuplot* for generering av grafiske fremstillinger av resultatdataene. *Gnuplot* kan lese kommandoer interaktivt eller direkte fra *standard in*. For å forenkle arbeidet med å generere grafer, har vi implementert et skript som genererer *Gnuplot*-instruksjoner for alle de forskjellige OCB-resultatdataene. Hele skriptet, `plot.pl`, er gjengitt i tillegg B.3.3. Eksempel på bruk av `plot.pl` samt resultat, er vist i programeksempel 5.3.3.

Utsnitt fra rapportfilen

```
1  ******************************
2  * Processor Move
3  ******************************
4
5                  Client  Server
6  Size: 16384
7          min     7       7
8          avrg    9       9
9          max     7029    6169
10
11 Size: 524288
12         min     2671    2778
13         avrg    3261    3373
14         max     12774   11235
```

Programeksempel 5.9: Resultat fra konvertering av maskinvare-resultater

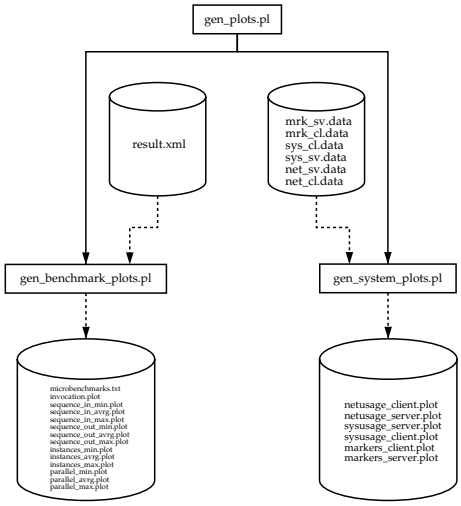
Wrapperskript

For hver ytelsesmåling genereres det oppunder 20 data-filer. For å forenkle arbeidet med å generere plot-filer av disse, har vi skrevet et *wrapper-skript* som genererer alle plot-filer for én ytelsesmåling (`gen_plots.pl`, se tillegg B.3.5). I tillegg har vi skrevet et tilsvarende skript for generering av postscript-versjoner av grafene (`gen_postscript.sh`, se B.3.4). Figur 5.2 viser hvilke skript som konverterer datafilene til plot-filer.

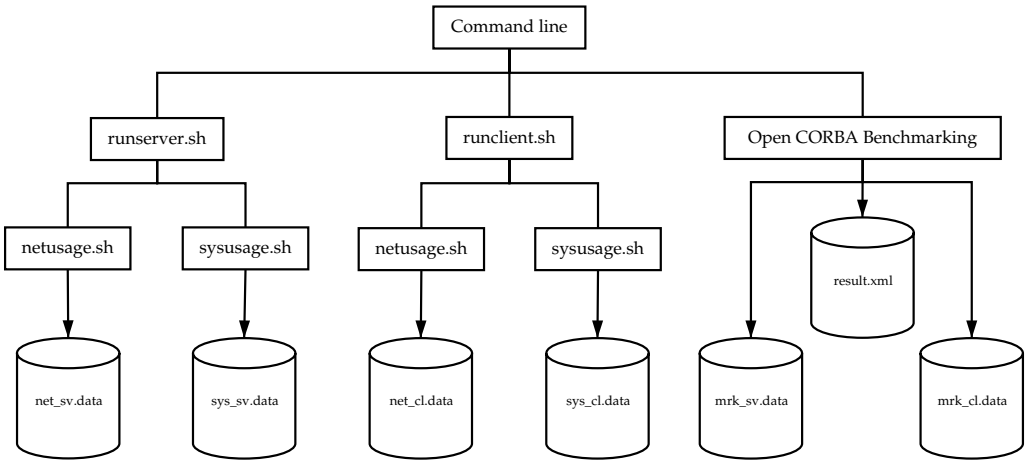
5.4 Kjøring av benchmark

Ved igangsetting av hver ytelsesmåling er det flere ting som må gjøres. For å forenkle denne operasjonen, skrev vi egne skript for start av tjener og klient. Skriptene tar et valgfritt argument, som videresendes til OCBs `Server`- eller `Client`-program. Vår modifiserte versjon av *Open CORBA Benchmarking* kan ta et `-nosys`-argument for å hoppe over de tidkrevende maskinvaretestene. Kildekoden til `runserver.sh` og `runclient.sh` finnes i henholdsvis tillegg B.1.1 og B.1.2. Programeksempel 5.4 viser start av ytelsesmåling ved hjelp av skriptet, og figur 5.3 viser sammenhengen mellom skriptene og hvilke datafiler som blir generert.

Det første skriptene gjør er å kjøre `ntpdate`. Dette synkroniserer systemklokken, slik at alle tidsangivelser i loggene for både klient og tjener



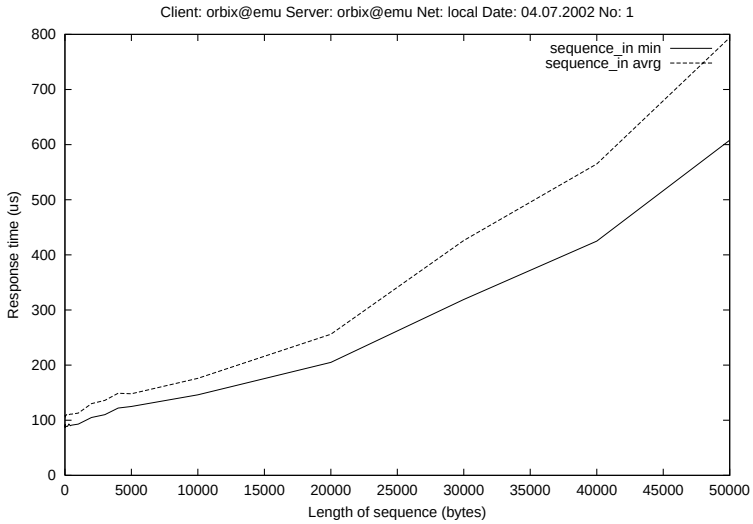
Figur 5.2: Visualisering av skriptene som konverterer data-filene til plot-filer.



Figur 5.3: Kjøring av benchmark

Eksempel på kommandolinje for plotting

```
plot.pl -n sequence_in -t min -t avrg
```



Programeksempel 5.10: Kommandolinje og tilsvarende resultat ved bruk av `plot.pl`.

Eksempel på starting av ytelsesmåling

```
$ LOGDIR=log/mico.ipaq-mico.qapi-WLAN-2002-07-01-1 \
  ./runserver -nosys &
```

Programeksempel 5.11: Eksempel på start av ytelsesmåling.

lar seg sammelikne. Deretter startes skriptet `netusage.sh` for logging av nettforbruk, etterfulgt av selve benchmark-programmet, `Client` eller `Server`. Prosessnummeret fra denne prosessen tas vare på og sendes med til neste skript, `sysusage.sh`, som måler prosessor- og minneforbruk. Skriptet venter på at denne prosessen skal avslutte, og avslutter så skriptet som logger nettforbruk.

For at test-objektet i ORB-en på klientsiden skal kunne kommunisere med tjenersidens test-objekt må det ha en referanse. Ved testing av MICO brukes en *Naming Service Daemon*. Denne kjører på den samme maskinen som loggene skrives til, og lytter på en bestemt port etter forespørsler (se programeksempel 5.12). Når tjeneren har opprettet et test-objekt og registrert det i den lokale ORB-en, kontakter den navnetjenesten for å registrere objektet. Klienten kan så ta kontakt på nevnte port og få en referanse til objektet.

Vi hadde problemer med å benytte navnetjeneren til Orbix/E og valgte i

Eksempel på bruk av MICOs NSD

```
$ ./Server -ORBInitRef \
  NameService=corbaloc::azrael.uio.no:12345/NameService
```

Programeksempel 5.12: Eksempel på bruk av MICOs NSD

stedet å valgte derfor å lagre objektreferansen, *Interoperable Object Reference* (IOR), for tjener-objektet til en fil. Denne ble så kopiert over til iPAQ-en før kjøring av `runclient.sh`.

For at logging av data skal kunne skje må partisjonen som loggene skrives til være NFS-montert på forhånd. For arbeidsstasjon 1 monteres den over Ethernet, mens for iPAQ-ene og arbeidsstasjon 2 skjer dette over Wi-Fi-forbindelsen.

5.5 Erfaring fra målinger

Under testkjøring av måleverktøyet opplevde vi at ikke alt fungerte slik vi hadde håpet. Enkelte problemer klarte vi å løse, mens andre forble uløste og begrenset dermed våre muligheter til samling av data til analysen.

Ved kjøring av Orbix/E over Wi-Fi opplevde vi at Open CORBA Benchmarking ikke fullførte. Da vi undersøkte dette nærmere fant vi at den under forskjellige testkjøringer feilet på ulike punkter i benchmarken. Videre feilsøking viste at den ved alle tilfellene feilet under overføring av store mengder data. Dette ble forsøkt løst ved å oppgradere *firmware*-en i Aironet-kortene til en nyere versjon, og da vi etter flere testkjøringer ikke klarte å reprodusere feilen, anså vi problemet som løst.

Flere av plot-filene som ble generert på grunnlag av ressursforbruket under ytelesemålingene inneholdt punkter som avvek kraftig fra de andre målepunktene. I de fleste tilfellene var verdien langt utenfor mulige maksimalverdier. Etter å ha studert dataene som ble logget fra `/proc/{PID}/stat`, fant vi at enkelte logglinjer inneholdt flere elementer enn det som er dokumentert i *man proc*. Man-sidene til *proc* ga ingen forklaring på hvorfor `stat`-filen av og til inneholder data som avviker fra det fastsatte formatet.

Skriptet som behandler data fra disse loggene, `gen_system_plots.pl`, krever at hver logglinje har samme format, og ettersom det kun var en relativt liten andel av logglinjene som inneholdt feil (se tabell 5.5), valgte

Utelukking av datalinjer med feil (gen_system_plots.pl B.3.1)

```
252         # Sometimes the data gets corruptet. Ignore it and
253         # log the event.
254         if ($#elems != 44) {
255             $errors++;
256             next();
257         }

322     # Log stats for amount of corrupt data.
323     warn(sprintf "$errors of $lines lines has errors (%3.2f\%).\n",
324          (($errors/$lines)*100));
```

Programeksempel 5.13: Rutine for gjenkjenning av og forkasting av feildata i systemforbruk-loggene.

Feil (%)	*Antall*
0	21
0,01	6
0,02 - 0,50	9
0,51 - 1,00	10
1,01 - 1,50	6
>1,51	0

Tabell 5.5: Feildata i logg fra sysusage. Se tabell A.3 i tillegget, for mer detaljer.

vi å forkaste disse. Programeksempel 5.13 viser rutinene for forkasting av feildata, samt loggerutinene som danner grunnlaget for tabell 5.5.

For MICO fungerte teknikken for logging av ressursforbruk smertefritt, mens vi for Orbix/E fikk større problemer. Loggingen av CPU-forbruk fungerte greit for klient-prosessen for en del av målingene, men ingen av tjener-prosessene hadde korrekt logging. I tillegg viste enkelte av målingene et minneforbruk som er langt høyere enn den totale mengden med fysisk og virtuelt minne på maskinen. Vi fant ikke årsaken til problemet og fikk derfor heller ikke skrevet om rutinene for logging av systemressursforbruken for Orbix/E-målingene.

For å studere effekten av systemressurslogging på målingene utførte vi for en testkonfigurasjon flere målinger: én hvor systemlogging foregikk som normalt (en måling hvert sekund), én hvor det gikk fem sekunder mellom hver måling, og én uten logging. Tabell 5.6 viser resultater for responstid

Måling	*Median*	*Gj.snitt*	*Logging*
23	4981	5870	Logging hvert sekund
26	4977	5454	Logging hvert 5. sekund
27	4963	5002	Ingen logging

Tabell 5.6: Responstid (i mikrosekunder) ved forskjellig grad av logging.

ved de respektive målingene, og vi ser her klare indikasjoner på at loggingen påvirker måledataene.

5.6 Sammendrag

Testene til *Open CORBA Benchmarking* måtte senke kravet til antall objekter under *dispatcher*-testene, ettersom iPAQ har begrenset med minne. Orbix/E hadde en grense på 1.000 simultane objekter, og dette måtte oppjusteres for å imøtekomme OCBs ønske om å opprette flere tusen objekter under enkelte av ytelsestestene.

Ettersom analyseverktøyet ikke ga oss all informasjonen vi har behov for, har vi skrevet egne verktøy for logging av ressursforbruk under ytelsesmålingene. Vi har også laget egne analyseverktøy for å analysere egne data, samt resultatene fra OCB-målingene. Loggeverktøyene har svakheter som vi var klar over når vi designet dem, men vi fant også svakheter etter at vi hadde utført ytelsesmålingene. Hovedsaklig er svakhetene relatert til loggeverktøyenes ressursforbruk på iPAQ.

Kapittel 6

Analyse

I dette kapittelet tar vi for oss analysen av de innsamlede data fra ytelsesmålingene med *Open CORBA Benchmarking*. Vi har totalt sett gjort rundt 40 målinger, hvorav 31 har blitt brukt som datagrunnlag for analysen. Tabell A.1 (side 144) i tillegget gir en komplett oversikt over disse.

Av forskjellige årsaker, omtalt avslutningsvis i forrige kapittel, har vi ikke fått gjort målinger for alle scenariene beskrevet i kapittel 3 (se 3.6, side 45). Vi har ikke brukt IrDA (scenario 13, 14 og 15) under noen av målingene, og heller ikke Wi-Fi i *AdHoc*-modus (scenario 6). I tillegg har vi behandlet Bluetooth-målinger konfigurert til å bruke henholdsvis *single slot*- og *5-slot*-pakketyper likt. Utover dette har vi ikke analysert parallellitetsresultater fra Orbix/E-målinger, da vi ikke har tilsvarende data fra MICO.

For metodekall-målinger har vi hatt tilgang til alle rådataene fra målingene (beskrevet i 4.1, side 49). Data fra *marshalling*- og *dispatcher*-målingene har dessverre vært begrenset til fem sett med verdier, bestående av én minimums-, maksimums- og gjennomsnittsverdi. Det statistiske grunnlaget har derfor vært for tynt til å beregne standardavvik og hoveddistribusjon for data fra disse målingene. Av denne grunn har vi hovedsakelig basert analysen på gjennomsnittet av de fem gjennomsnittsverdiene. Minimums- og maksimumsverdiene har kun vært brukt til å belyse enkelte tendenser ved gjennomsnittsverdiene. I de tilfellene hvor vi har gjort flere målinger med identisk testoppsett, har vi i de fleste tilfeller brukt gjennomsnittet av disse. I tabellene vises denne sammenslåingen ved bruk av tegnet '&' mellom målingsnumrene.

Under marshalling- og dispatcher-målingene ble det samlet data for hen-

holdsvis ulik datastørrelse og antall objekter. Under analysen av disse målingene har vi stort sett benyttet verdiene for ytterpunktene. For marshalling vil dette si responstid-verdier for datastørrelse på 0 KB og 50 KB, mens det for dispatcher vil være 1 objekt og 20.000 objekter. Disse verdiene er brukt til å beregne økning i responstid, samt forholdstall. I de tilfelle hvor utviklingen ikke har vært tilnærmet lineær har vi brukt mellomliggende data, samt grafiske plot til å vise tendenser. For metodekallmålinger har det ved plotting av forekomster vært nødvendig å vise et utvalg sentrert rundt hoveddistribusjonen grunnet stor spredning av måleforekomster. Utvalget er da basert på utregninger av kvartiler (omtalt i tabellene som *Q1*, *Q3* og *median*).

Analysen av målingene er delt inn i tre hoveddeler basert på ulik fokus. Den første delen tar for seg de to CORBA-implementasjonene, Orbix/E og MICO. Del to fokuserer på maskinvarens påvirkning, og sammenlikner kjøring på iPAQ og arbeidsstasjon. Her ser vi også på hvordan ulike klient-tjener-kombinasjoner påvirker ytelsen. I den siste delen analyserer vi bruk av to forskjellige nettverksteknologier, Wi-Fi og Bluetooth, og setter resultatene opp målinger gjort uten nettverk. Hver av disse tre hoveddelene har samme struktur: vi ser først på *metodekall*, deretter *marshalling*, og til slutt *dispatcher*.

Avslutningsvis følger en analyse av systemressursforbruket, som kan belyse noen av resultatene presentert i de foregående delkapittelene. Deretter gir vi en kort presentasjon av andres resultater, og runder av med et kort sammendrag av våre egne resultater.

	Maskin		Responstid Snitt	Q1	Median	Q3
MICO	ipaq	(4)	6.237	4.325	4.361	4.406
	pc1	(3)	732	648	659	663
	pc2	(6&24)	355	324	325	327
Orbix/E	ipaq	(31)	1.627	1.156	1.206	1.208
	pc1	(1&28)	237	209	211	213
	pc2	(7)	107	90	90	91

Tabell 6.1: Responstider for MICO og Orbix/E på forskjellige maskiner (gjennomsnittsverdier er brukt for pc1 og pc2 med Orbix/E).

6.1 CORBA-implementasjon

I dette delkapittelet presenterer vi sammenlikninger av resultatene fra de forskjellige hovedkategoriene i OCB-målingene, foretatt lokalt på iPAQ-en og de to arbeidsstasjonene. Responstidene for MICO og Orbix/E er målt ved å kjøre både klient og tjener på samme maskin. Resultatet fra disse målingene belyser ulikhetene mellom CORBA-implementasjonene uten at faktorer som nettverksforhold eller maskinvarekonfigurasjoner spiller inn.

6.1.1 Metodekall

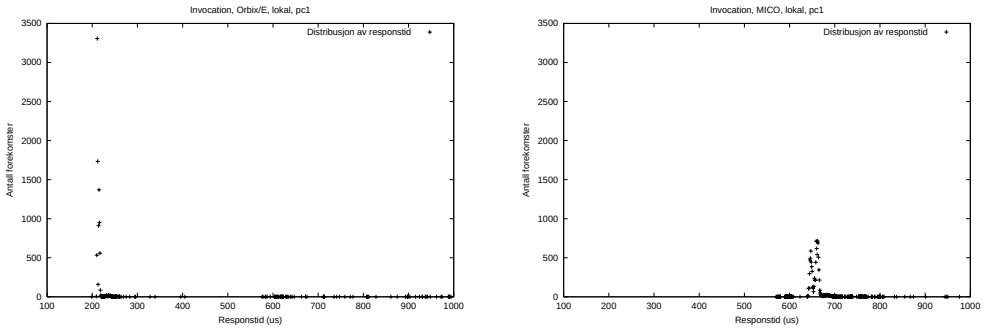
Figur 6.1 gir en oversikt over responstider for MICO og Orbix/E på forskjellige testmaskiner. For hver av sammenlikningene mellom Orbix/E og MICO på de tre testmaskinene har vi regnet ut forholdstall for responstid mellom CORBA-implementasjonene. Ved å dele responstid for Orbix/E ($rt_{Orbix/E}$) med de respektive verdier for MICO (rt_{MICO}), fikk vi et forholdstall for responstiden til ORB-ene. Likning 6.1 ble brukt til å beregne dette forholdstallet. Dette ble gjort både med gjennomsnittlig responstid og medianen, og resultatene fra utregningene er presentert i tabell 6.2. I de tilfellene hvor vi hadde flere målinger for helt like konfigurasjoner brukte vi gjennomsnittet av disse.

$$forholdstall_{6.1} = \frac{rt_{Orbix/E}}{rt_{MICO}} \tag{6.1}$$

Maskin		$Forholdstall_{6.1}$ *Snitt*	*Median*
ipaq	(31,4)	0,28	0,26
pc1	(1&28,3)	0,32	0,32
pc2	(7,6&24)	0,30	0,27

Tabell 6.2: Forholdstall for gjennomsnittlig responstid og median mellom Orbix/E og MICO uten bruk av nettverk, beregnet med likning 6.1.

Resultatene i tabell 6.1 viser tydelig at MICOs responstider ligger godt over Orbix/Es for alle maskinene. Forholdet mellom de to CORBA-implementasjonene blir tydeligere gjennom utregningene av forholdstall (vist i tabell 6.2). MICO bruker jevnt over tre ganger så lang tid på å svare som Orbix/E.



(a) Orbix/E (1)

(b) MICO (3)

Figur 6.1: *Invocation* for Orbix/E og MICO.

Figur 6.1 viser et eksempel på fordeling av responstid for Orbix/E og MICO. Av tabell 6.1 kan vi se at hoveddistribusjonen for Orbix/E ligger på 209-213μs, mot 648-663μs for MICO. Plottene i figuren viser det samme; Orbix/E har lavere responstider og noe mindre spredning i responstidene enn MICO. Merk at tallene for Orbix/E er gjennomsnittsverdier fra måling 1 og 28, mens figuren viser plot av måling 1 (se tabell A.2 på side 145 i tillegget for de individuelle resultatene).

Undersøkelsene våre viser at responstidene til Orbix/E ligger godt under MICOs. Forholdet mellom de to CORBA-implementasjonene ligger på omkring en tredjedel for lokale målinger.

	Maskin	ORB	Responstid 0 KB	Responstid 50 KB	Økning	
In	pc1 (1)	Orbix/E	243μs	1.262μs	1.019μs	419%
	pc1 (3)	MICO	789μs	2.785μs	1.996μs	253%
	pc2 (7)	Orbix/E	107μs	794μs	687μs	642%
	pc2 (6)	MICO	391μs	1.986μs	1.595μs	408%
Out	pc1 (1)	Orbix/E	270μs	1.604μs	1.334μs	494%
	pc1 (3)	MICO	795μs	4.312μs	3.517μs	442%
	pc2 (7)	Orbix/E	119μs	1.090μs	971μs	816%
	pc2 (6)	MICO	402μs	3.007μs	2.605μs	648%

Tabell 6.3: Data fra *Sequence in* og *out* for MICO (3, 6) og Orbix/E (1, 7)

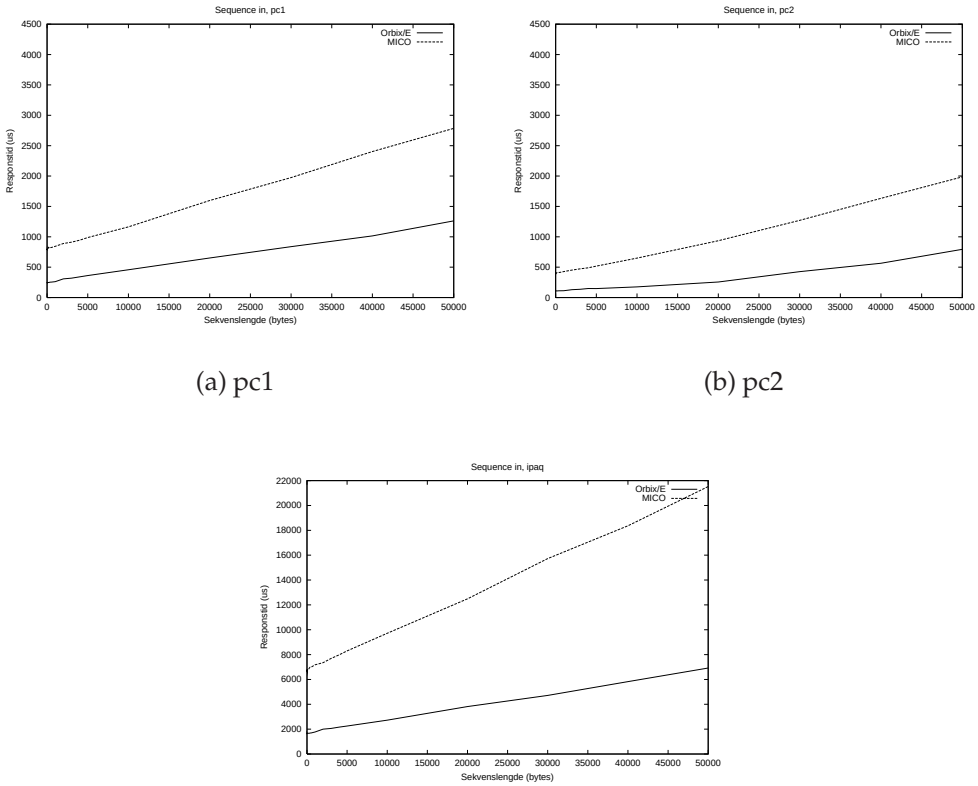
	Maskin		$Forholdstall_{6.1}$ 0 KB	$Forholdstall_{6.1}$ 50 KB
In	pc1	(1,3)	0,31	0,45
	pc2	(7,6)	0,27	0,40
	ipaq	(31,4)	0,25	0,32
Out	pc1	(1,3)	0,34	0,37
	pc2	(7,6)	0,30	0,36
	ipaq	(31,4)	0,28	0,28

Tabell 6.4: Forholdstall mellom Orbix/E og MICO for *Sequence in* og *out*, beregnet med likning 6.1, side 81.

6.1.2 Marshalling

For å studere hvordan responstidene varierer med økende mengde parameter- og returdata, har vi beregnet den prosentvise økningen av responstiden mellom metodekall med og uten data. Metodekall gjort uten dataoverføring tilsvarer *invocation*-testen, som ble omtalt i forrige avsnitt.

For at resultatene ikke skal påvirkes av forsinkelse i nettverket har vi sammenliknet data fra lokale målinger. Tabell 6.3 viser responstid for parameterstørrelse på 0 og 50 KB for både *sequence in* og *sequence out*. I tillegg viser tabellen den absolutte økningen i responstid fra kall uten parameterdata til 50 KB datamengde. Tabell 6.4 viser forholdet mellom responstider for Orbix/E og MICO for parameterstørrelse 0 KB og 50 KB.



(a) pc1

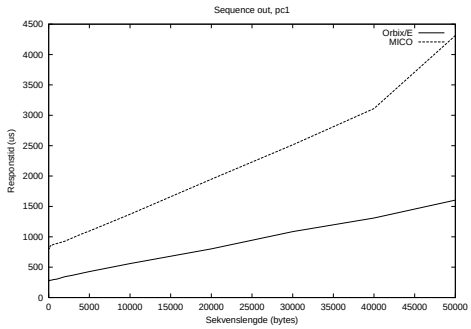
(b) pc2

(c) ipaq

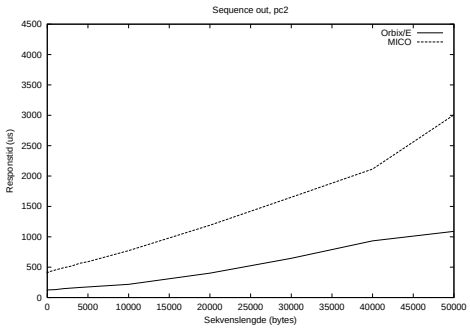
Figur 6.2: *Sequence in* for Orbix/E (1, 31 og 7) og MICO (3, 4 og 6).
Merk: Skalaene er ikke like for alle grafene.

Som også *invocation*-testen viste, er Orbix/E markant raskere enn MICO. På begge testmaskinene brukte Orbix/E rundt en tredjedel av responstiden til MICO for målinger uten dataoverføring. Med overføring av 50 KB får implementasjonen av marshalling-/demarshalling-rutinene større innvirkning, og her viser målingene at Orbix/E bruker 40-45% av tiden MICO bruker på å svare (se tabell 6.4). Av tallene for absolutt økning ser vi også at MICO gir dårligst resultat. Fra 0 til 50 KB parameterstørrelse øker MICOs responstid omtrent dobbelt så mye som Orbix/Es. Figur 6.2 viser alle resultatene fra *sequence in*-målingene gjort mot Orbix/E og MICO.

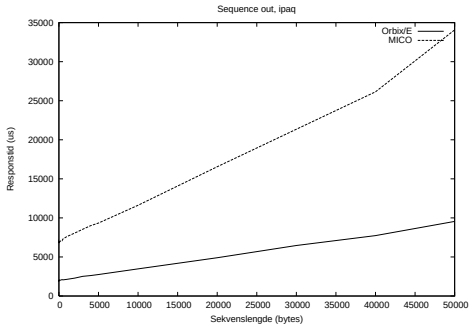
Sequence out viser samme tendens som *sequence in*. Også her starter Orbix/Es responstider lavere enn MICOs. Responstidene for begge CORBA-implementasjonene er her noe høyere enn for *sequence in*, mens forskjellen



(a) pc1



(b) pc2



(c) ipaq

Figur 6.3: *Sequence out* Orbix/E (1, 31 og 7) og MICO (3, 4 og 6).
Merk: Skalaene er ikke like for alle grafene.

mellom dem er noe mindre (se figur 6.3). Videre ser vi av tabell 6.3 at økningen i responstid for begge implementasjonene er større for *sequence out* enn *sequence in*.

Begge CORBA-implementasjonene har lengre responstid på *sequence out*- enn *sequence in*-testene. Tabell 6.5 viser den prosentvise forskjellen mellom de to sekvens-testene. Likning 6.2 er brukt til disse utregningene.

$$forhold_{6.2} = \frac{rt_{sequence\_in}}{rt_{sequence\_out}} \times 100 \tag{6.2}$$

Vi ser av tabellen at resultatene er relativt like for begge maskinene: ved tomt metodekall utgjør responstiden ved *sequence in* 90% av responstiden

	ORB		$Forhold_{6.2}$ 0 KB	50 KB
pc1	Orbix/E	(1)	90%	79%
	MICO	(3)	99%	65%
pc2	Orbix/E	(6)	90%	73%
	MICO	(7)	97%	66%

Tabell 6.5: Forholdet mellom *Sequence in* og *out*-testen, beregnet med likning 6.2.

ved *sequence in* for Orbix/E og 97-99% for MICO. Ved overføring av 50 KB er heller ikke forskjellene store; verdiene for *sequence in* utgjør rundt tre fjerdedeler og to tredjedeler av *sequence out*-verdiene for henholdsvis Orbix/E og MICO. Vi har ikke kartlagt årsaken til at den ene sekvenstesten bruker lengre tid enn den andre.

MICO er markant tregere enn Orbix/E både ved sending av parametere og returverdier. I tillegg skalerer MICO dårligere enn Orbix/E for begge sekvenstestene, spesielt *sequence out*, noe som vises ved at i MICOs grafer stiger raskere enn Orbix/Es. Dette tyder på at MICOs IDL-kompilator genererer mindre effektiv kode enn IDL-kompilatoren til Orbix/E. Målingene viser en klar forskjell mellom *sequence in* og *sequence out*-testene; ved store datamengder er det markant mer ressurskrevende å returnere data enn å overføre dem som parametre til et metodekall.

6.1.3 Dispatcher

For å kunne studere forskjellen i responstid for ulikt antall objekter på tjenersiden gjorde vi lokale målinger på arbeidsstasjon og iPAQ for begge CORBA-implementasjonene. Grunnet problemer med Orbix/E på iPAQ, omtalt i delkapittel 5.1.1 (side 58), ble det for dette test-scenariet ikke målt responstid for mer enn 10.000 objekter. For å kunne sammenlikne med CORBA-implementasjonene ble derfor resultatene fra målingen med 20.000 objekter i MICO utelatt.

Fra de innsamlede dataene regnet vi ut økning i responstid i mikrosekunder, samt den prosentvise økningen. I tillegg ble forholdet mellom responstiden for henholdsvis Orbix/E og MICO regnet ut basert på likning 6.1. Tabell 6.6 og 6.7 viser resultatet av disse utregningene.

Maskin	*ORB*	*Responstid* 1 obj	10000 obj	*Økning*	
pc1 (28)	Orbix/E	246μs	252μs	6μs	2%
pc1 (3)	MICO	739μs	817μs	78μs	11%
ipaq (31)	Orbix/E	1.685μs	1.961μs	276μs	16%
ipaq (4)	MICO	6.343μs	9.507μs	3.164μs	50%

Tabell 6.6: Data fra *instances* for Orbix/E (28,31) og MICO (3,4).

Maskin	$Forhold_{6.3}$ 1 obj	10000 obj
pc1	33%	31%
ipaq	27%	21%

Tabell 6.7: Forholdet for *instances* mellom Orbix/E og MICO.

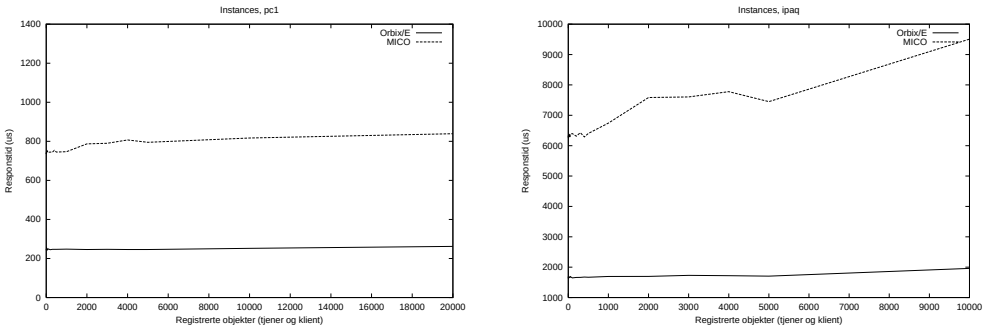
Vi ser av tabell 6.6 at MICO har høyere responstider enn Orbix/E både ved 1 og 10.000 objekter. Dette gjelder både ved kjøring på arbeidsstasjon og iPAQ. Økningen i responstid ved økende antall objekter er også er høyere for MICO. Dette gjelder både for absolutt og prosentvis økning.

$$forhold_{6.3} = \frac{rt_{Orbix/E}}{rt_{MICO}} \tag{6.3}$$

Resultatene i tabell 6.7 forteller oss at responstiden til Orbix/E på arbeidstasjonen ligger på 33% av MICOs responstid for 1 objekt. Ved økning til 10.000 objekter synker dette tallet med 2 prosentpoeng. På iPAQ responstiden til Orbix/E på 27% av MICOs for 1 objekt, og 6 prosentpoeng lavere 10.000. Selv om det er store forskjeller i både den absolutte og prosentvise økningen i responstid for hver av de to CORBA-implementasjonene, gir altså ikke dette de helt store utslagene på forholdet mellom responstidene.

Figur 6.4 viser skaleringstesten på to forskjellige maskiner. Vi ser her ganske klart det dataene og beregningene indikerer: uavhengig av hvilken maskin målingene blir kjørt på har Orbix/E lavere responstider enn MICO, og skalerer også bedre ved økende antall objekter.

Den gjennomgående tendensen er at Orbix/E er raskere enn MICO også under *instance*-testene. I tillegg skalerer Orbix/E bedre enn MICO. Dog er forholdet mellom responstider rimelig jevnt hele veien. Resultatene kan



(a) pc1 (28, 3)

(b) ipaq (31, 4)

Figur 6.4: *Instances* for Orbix/E og MICO på pc1 og ipaq.

derfor tyde på at *dispatcheren* i Orbix/E er mer effektivt implementert enn i MICO.

6.1.4 Oppsummering

Alle *Open CORBA Benchmarking*-målingene indikerer at MICO har lengre responstid enn Orbix/E. Enkle *metodekall*-tester viser at gjennomsnittlig responstid for Orbix/E ligger på en tredjedel av tilsvarende verdier for MICO. Vi fant også at Orbix/E skalerer bedre både ved økende parameter- og returdatastørrelse og antall objekter på tjenersiden. Ved økende parameterstørrelse minsker forholdstallet for de to CORBA-implementasjonene, mens det ved økende antall objekter øker.

Alle sekvenstestene viser at responstiden ved retur av parameterdata er høyere enn sending av parameterdata. Dette gjelder begge ORB-ene og vi er usikre på om dette skyldes implementasjonen av analyseverktøyet eller om årsaken ligger i CORBA-arkitekturen.

	Prosessor			*Minne*	*Tråder*		
Maskin	*Move*	*Int*	*Float*	*alloc*	*Life*	*Switch*	*Lock*
ipaq	17.655μs	346μs	2.783μs	1.103μs	10.8314μs	2.088μs	153μs
pc1	3.625μs	208μs	229μs	417μs	39.333μs	2.078μs	61μs
pc2	3.130μs	141μs	156μs	276μs	21.167μs	1.507μs	40μs

Tabell 6.8: Microbenchmarks for pc1, pc2 og ipaq.

6.2 Maskinvare

Dette delkapittelet omhandler målinger gjort med forskjellige maskinvarekonfigurasjoner. Vi har benyttet ulik maskinvare, både ytelses- og arkitekturmessig, og sammenliknet resultater fra målinger gjort lokalt, men også resultater for forskjellige kombinasjoner av maskinvare. Ideelt skulle de sistnevnte målinger vært gjort uten bruk av nettverk, for å hindre forsinkelsen nettverkslaget innfører, men dette er naturlig nok ikke mulig. Vi har derfor hovedsakelig basert oss på den nettverksteknologien vi har maskinvare til, med lavest latens og størst gjennomstrømning: Wi-Fi. *Open CORBA Benchmarking* har egne ytelsestester for maskinvaren, som vi vil bruke for å belyse resultatene av CORBA-målingene. Resultater fra disse målingene presenteres i første delkapittel.

6.2.1 Maskinvareytelse

Open CORBA Benchmarking foretar målinger (*microbenchmarks*) av maskinvarens ytelse før den starter målingen av CORBA-implementasjonen. Nærmere beskrivelse av målemetodene finnes i delkapittel 4.1.1 (side 50). Snittresultatene fra et utvalg av målingene gjort på ipaq og de to stasjonære maskinene, er presentert i tabell 6.8. Størrelsen for *processor move* er 512 kilobytes, minneallokeringen er repetert 100 ganger med en størrelse på 1536 bytes og *thread life* er målt ved bruk av 128 tråder.

For å vise forskjellen mellom de forskjellige maskinene, har vi satt opp en tabell (6.9) med sammenlikning av alle maskinene. Likning 6.4 er benyttet til å beregne forholdet i mellom de forskjellige maskinene og resultatene viser at pc2 har lavere responstid enn pc1 for alle testene, rundt regnet 2:3. Begge de stasjonære maskinene har høyere ytelse enn ipaq under alle testene, men enkelte av målingene skiller seg ut. Forskjellen mellom i386- og ARM-arkitekturen viser seg med at resultatene fra heltalls-testen er mind-

Maskin		*Prosessor*			*Minne*	*Tråder*		
1	2	*Move*	*Int*	*Float*	*alloc*	*Life*	*Switch*	*Lock*
pc2	pc1	86%	68%	68%	66%	54%	73%	66%
pc2	ipaq	18%	41%	6%	25%	20%	72%	26%
pc1	ipaq	21%	60%	8%	38%	36%	99%	40%

Tabell 6.9: Forhold mellom pc1, pc2 og ipaq for microbenchmark-målinger, beregnet med likning 6.4.

re enn de andre testene (40-60%). Flyttallsoperasjoner er derimot langt raskere på i386; De stasjonære maskinene bruker 6-8% av responstiden til ipaq. iPAQ er ikke utstyrt med flyttallsprosessor (FPU), i motsetning til de Pentium-prosessorene i de stasjonære maskinene, og dette forklarer den kraftige forskjellen i ytelse tilknyttet flyttallsoperasjonene. For trådbytte er det derimot lite forskjell mellom arkitekturene, spesielt mellom pc1 og ipaq hvor responstiden er tilnærmet lik hverandre.

$$forhold_{6.4} = \frac{resultat_{maskin1}}{resultat_{maskin2}} \times 100\% \tag{6.4}$$

Som nevnt er ipaq utstyrt med en Intel StrongARM, en RISC[1] prosessor, mens de to stasjonære maskinene er utstyrt med Pentium-prosessorer. Sistnevnte er basert på CISC[2]-prosessor. CISC- og RISC-arkitekturene er grunnleggende forskjellig og ettersom vi har benyttet samme Linux-kjerne (2.4.18) for alle målingene, er det nærliggende å forklare sammenhengen mellom forholdstallene (i tabell 6.9) med forskjellen i prosessorarkitekturen eller de arkitekturspesifikke delene av operativsystemkjernen.

6.2.2 Metodekall

På samme måte som for CORBA-analysen har vi brukt *invocation*-testen i *Open CORBA Benchmarking* for å måle responstiden for enkle metodekall. Målingene ble gjort mot begge ORB-ene lokalt, på alle maskinene, noe som gir oss resultater som ikke er påvirket av nettverkslaget. Nøkkeltall, som beskriver distribusjonen av responstidene, er presentert i tabell 6.10. Som vi har sett av tidligere målinger er det en markant forskjell mellom pc1 og pc2; pc2 har omtrent halve responstiden av pc1. iPAQ-en er som ventet

[1] Reduced Instruction Set Computer
[2] Complex Instruction Set Computer

Maskin	*Snitt*	*Q1*	*Median*	*Q2*
pc1 (1)	236μs	211μs	212μs	215μs
pc2 (7)	107μs	90μs	90μs	91μs
ipaq (31)	1.627μs	1.156μs	1.206μs	1.208μs

Tabell 6.10: Responstider for Orbix/E (*invocation*) lokalt på pc1, pc2 og ipaq.

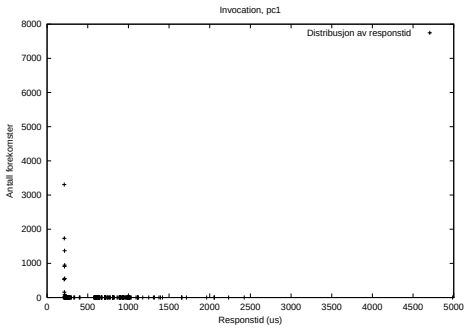
også langt tregere enn de to stasjonære maskinene og pc1 bruker kun 6-8% av ipaqs responstid. Dette stemmer overens med resultatene fra maskinvaretestene i *Open CORBA Benchmarking* (tabell 6.8).

Figur 6.5 viser utsnitt av plottene for målingene på de tre maskinene. Det kan være vanskelig å se distribusjonen av plotpunktene grunnet skalaen til y-aksen, men dette skyldes at pc2 har over 7.000 forekomster av samme målresultat (90μs). pc1 har ikke fullt så liten spredning, men fortsatt mindre enn ipaqs. Dette tyder på at spredningen er avhengig av ytelsen til maskinen.

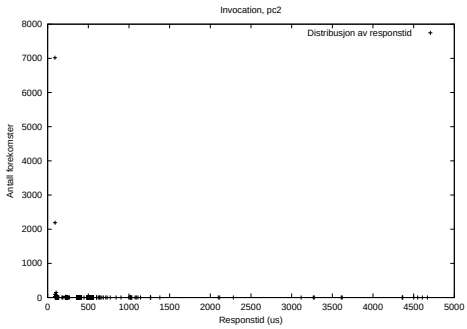
De overnevnte målingene ble gjort lokalt på samme maskin, men vi ønsket også å se om forskjellige maskinvarekonfigurasjoner for tjener og klient har innvirkning på måleresultatene. Vi satte derfor opp kombinasjoner med maskiner, alle med ipaq, ved hjelp av Wi-Fi-nettverk. De høye gjennomsnittsverdiene, i forhold til medianen, viser at responstiden for en del av målingene har ligget langt utenfor hoveddistribusjonen. Det er tvilsom at dette skyldes valg av maskinvarekombinasjon og skyldes mest sannsynlig introduksjonen av Wi-Fi-nettverksforbindelsen.

Kombinasjonen med de dårligste responstidene var ipaq-ipaq2 (se tabell 6.11 og figur 6.6). Totalt sett var det kombinasjonen med minst samlet prosessorkraft, og resultatet var derfor som forventet. Med Orbix/E ga kombinasjonene ipaq-pc1 og pc1-ipaq omtrent samme responstid og spredning av målingene er tilnærmet lik.

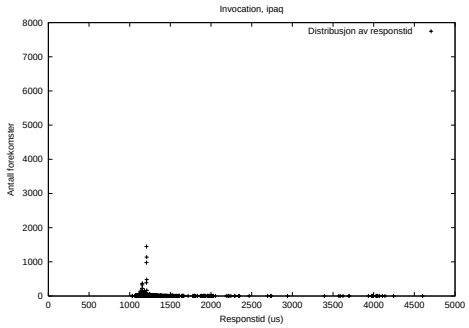
Kombinasjonene med pc1 og ipaq ga et noe uventet utslag: For både MICO og Orbix/E er distribusjonen av målingene relativt lik for begge klient-tjener-kombinasjoner, men kombinasjonen med ipaq som tjener resulterte i responstider som i snitt ligger omtrent tusen mikrosekunder høyere (20%). Figur 6.7 viser tydelig denne forskjellen. Dette indikerer at tjeneren krever mer prosessorkraft enn klienten og at den beste kombinasjonen derfor er å benytte den mest ressurssterke maskinen på tjenersiden.



(a) pc1 (1)



(b) pc2 (7)



(c) ipaq (31)

Figur 6.5: *Invocation* for Orbix/E lokalt på pc1, pc2 og ipaq

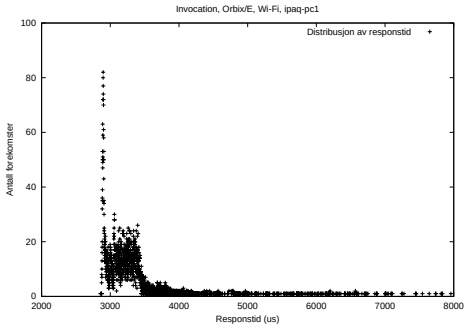
I forbindelse med nettverksanalysen gjorde vi responstidmålinger uten bruk av CORBA ved forskjellige maskinvarekombinasjoner. Resultatene er presentert i figur 6.20 (side 108), og forteller oss at at maskinkombinasjonen pc1-ipaq gir lavere responstider enn ipaq-pc1 over Bluetooth. Resultatene fra Wi-Fi-målingene er dessverre ikke nøyaktige nok til at vi kan si sikkert hvilken kombinasjon som gir best resultat. For å belyse denne problemstillingen nærmere, studerte vi resultatene fra *Open CORBA Benchmarking*s egne nettverksmålinger. For iPAQ varierer måleresultatene kraftig og det er ikke samsvar mellom målinger gjort på klient og tjener. Vi konkluderte med at *OCBs* rutiner for *socket ping*-test inneholder feil som resulterer i feil måledata under kjøring på ARM-prosessoren.

Ikke uventet så vi altså en klar forskjell i responstidene på de forskjellige maskinene. De to stasjonære maskinene er langt raskere enn iPAQ-en. Det-

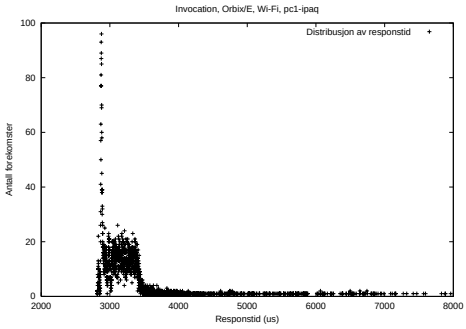
	Maskiner			*Snitt*	*Q1*	*Median*	*Q3*
Orbix/E	ipaq	pc1	(10)	3.734μs	2.981μs	3.180μs	3.358μs
	pc1	ipaq	(25)	3.676μs	2.932μs	3.132μs	3.311μs
	ipaq	ipaq2	(14)	5.698μs	4.071μs	4.265μs	4.474μs
MICO	ipaq	pc1	(9)	4.979μs	4.025μs	4.242μs	4.421μs
	pc1	ipaq	(23)	5.870μs	4.785μs	4.981μs	5.165μs
	ipaq	ipaq2	(13)	8.914μs	6.647μs	6.845μs	7.081μs

Tabell 6.11: Data fra *invocation* for MICO og Orbix/E

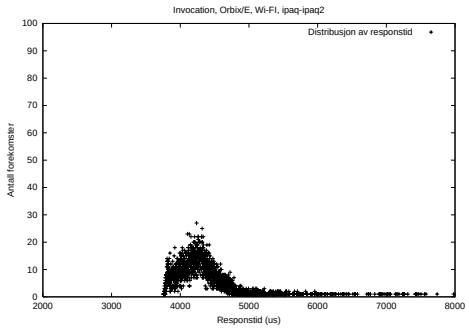
te stemmer også godt overens med spesifikasjonene av maskinvaren (se delkapittel 4.1.1, side 50) og OCBs *microbenchmarks* (tabell 6.8). Ved kombinasjon av maskiner er responstiden for MICO best når den stasjonære maskinen står som tjener. For Orbix/E har kombinasjonene ipaq-pc1 og pc1-ipaq tilnærmet like lang responstid — ipaq som tjener gir marginalt lavere responstid. Det er naturlig å anta at det kreves mer ressurser av en tjener enn en klient og vi antok derfor at den raskeste kombinasjonen alltid ville være med pc1 som tjener, og har ikke funnet noen forklaring på hvorfor dette ikke er tilfellet for Orbix/E.



(a) ipaq - pc1 (10)

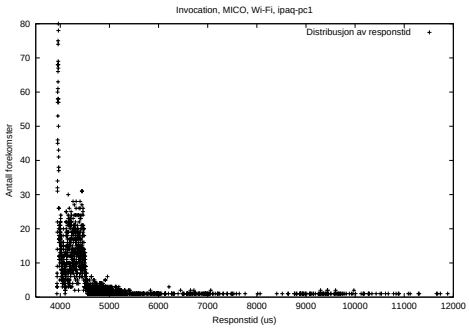


(b) pc1 - ipaq (25)

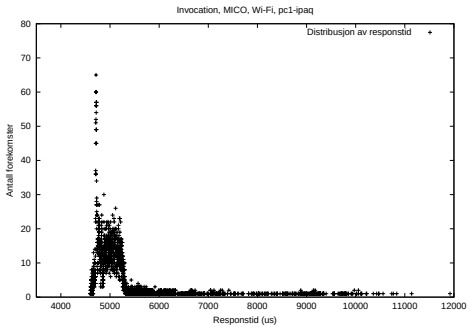


(c) ipaq - ipaq2 (14)

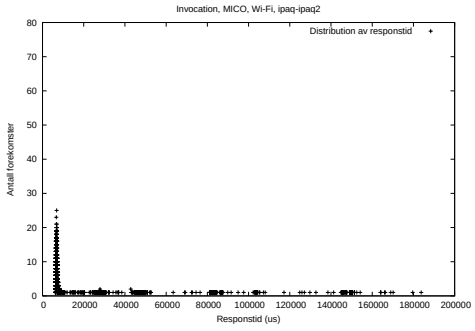
Figur 6.6: *Invocation* ipaq-pc1, pc1-ipaq og ipaq-ipaq2 med Orbix/E over Wi-Fi



(a) ipaq - pc1 (9)



(b) pc1 - ipaq (23)



(c) ipaq - ipaq2 (13)

Figur 6.7: *Invocation* for ipaq-pc1 og pc1-ipaq med MICO over Wi-Fi

6.2.3 Marshalling

Som beskrevet i delkapittel 6.1.2 viser ytelsesmålingene at returnering av data (*sequence out*) bruker lengre tid og skalerer dårligere enn sending av parameterdata. Den samme tendensen vises for målinger gjort på alle maskinene; Se tabell 6.12, som viser forholdene mellom *sequence in* og *out*, utregnet med likning 6.5. Alle maskinene har lengre responstid på *sequence out*-testen enn *sequence in*-testen både ved overføring av minimal og maksimal datamengde. Forholdet mellom de to testene er også relativt likt for ipaq og pc2, men for pc1 er det derimot mindre forskjell. Vi har ikke funnet noen forklaring på dette og det er muligens tilfeldig at den raskeste og tregeste maskinen skiller seg fra den tredje maskinen.

$$forhold_{6.5} = \frac{rt_{sequenceout}}{rt_{sequencein}} \times 100\% \tag{6.5}$$

Ved å sammenlikne resultatene for alle maskinene mot hverandre, ser vi hvor store utslag maskinvaren forårsaker. Tabell 6.13 inneholder forholdstallene for målinger gjort på forskjellige maskiner (likning 6.6) og disse viser at pc2 utfører *sequence*-testene 30-50% raskere enn pc1. iPAQ-en er cirka 10-15 ganger tregere enn pc-ene. Det er med andre ord betydelige forskjeller i ytelsen mellom iPAQ-en og de stasjonære maskinene.

$$forholdstall_{6.6} = \frac{resultat_{kolonne2}}{resultat_{kolonne1}} \tag{6.6}$$

For studere forskjeller i kombinasjoner av maskiner satte vi opp testkonfigurasjoner på følgende måte: pc1-ipaq, ipaq-ipaq2 og ipaq-pc1. Vi ønsket å minimere påvirkningen av nettverket og det beste hadde vært å benyttet 100 Mbit Ethernet under målingene. Ettersom vi ikke har maskinvare for dette, valgte vi å bruke Wi-Fi-nettverk i stedet, da dette er den av de tilgjengelige nettverksteknologiene som har lavest latens og størst båndbredde. Forholdstallene (tabell 6.15) viser at ved metodekall uten parameterdata, er forskjellen minimal mellom de to sekvenstestene. Ved overføring av 50 kilobytes med data er forskjellene større. Kombinasjonen ipaq-pc1 skiller seg ut ved at det tar lengre tid å sende parameter- enn returdata, noe som skyldes at iPAQ-en er tregere enn pc-en og at denne forskjellen overskygger forskjellen mellom ressursforbruket til de to testene — PC-en klarer å returnere data raskere enn iPAQ-en klarer å sende.

	$Forhold_{6.5}$	
Maskin	*0 KB*	*50 KB*
pc1 (3)	86%	79%
pc2 (6)	96%	66%
ipaq (4)	97%	63%

Tabell 6.12: Forhold mellom responstid for *sequence in* og *sequence out* mot MICO lokalt på pc1, pc2 og ipaq, basert på likning 6.5.

			$Forhold_{6.6}$	
	Maskiner		*0 KB*	*50 KB*
In	pc2	pc1	50%	71%
	pc2	ipaq	6%	9%
	pc1	ipaq	12%	13%
Out	pc2	pc1	51%	70%
	pc2	ipaq	6%	9%
	pc1	ipaq	12%	13%

Tabell 6.13: Forhold for *Sequence in* og *out* mellom pc1 (3), pc2 (6) og ipaq (4), beregnet med likning 6.6, side 96.

Tabell 6.14 inneholder måledata for sending/returnering av kall uten data og med 50 kilobytes for de forskjellige testkombinasjonene. Den minste stigningen i responstid får man med kombinasjonen hvor pc-en står for sending/returnering av data. Med andre ord er den beste kombinasjonen for *sequence in* testen at pc-en er klient, mens for *sequence out* er det mest effektivt om pc-en er tjener. Operasjoner som innebærer sending av data er altså tyngre enn operasjonene som mottar data. Vi er usikre på om dette skyldes maskinvaren vi har valgt, designen av ORB-ene eller underliggende nettverkslag. På TCP-nivå er den tyngste operasjonen å motta data, noe som blant annet skyldes sammensetning av defragmentering[36]. Dette er motsatt av hva resultatene våre viser, noe som tyder på at årsaken til ytelsesforskjellen mellom sending og mottak av data ligger på et annet nivå enn TCP-laget.

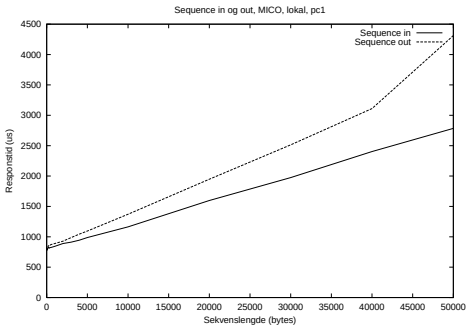
Målingene som ble gjort over Wi-Fi ga resultater som viser at den relative forskjellen mellom sekvenstestene er mindre enn for de lokale målingene (se figur 6.9 og 6.8). Dette kan forklares med at innføringen av Wi-Fi-nettverk medfører en del forsinkelse som overskygger en del av forskjellene i prosesseringstiden.

	Maskiner			*Responstid* 0KB	50KB	*Økning*	
In	pc1	ipaq	(23)	6.107μs	10.2820μs	96.713μs	1.583%
	ipaq	pc1	(9)	5.221μs	11.7148μs	111.927μs	2.144%
	ipaq	ipaq2	(13)	9.494μs	18.5571μs	176.077μs	1.855%
Out	pc1	ipaq	(23)	6.198μs	13.1284μs	125.086μs	2.018%
	ipaq	pc1	(9)	5.310μs	10.2569μs	97.259μs	1.832%
	ipaq	ipaq2	(13)	9.454μs	19.8528μs	189.074μs	2.000%

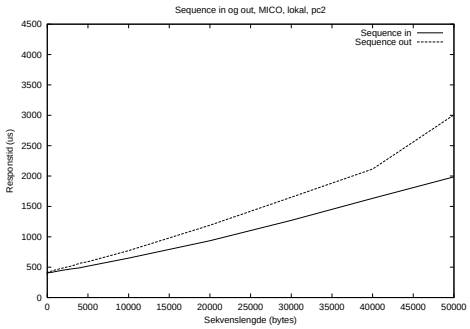
Tabell 6.14: Data fra *Sequence in* og *out* for MICO på pc1-ipaq, ipaq-pc1 og ipaq-ipaq2.

Maskiner			$Forhold_{6.5}$ 0 KB	50 KB
pc1	ipaq	(23)	98%	78%
ipaq	pc1	(9)	98%	114%
ipaq	ipaq2	(13)	98%	93%

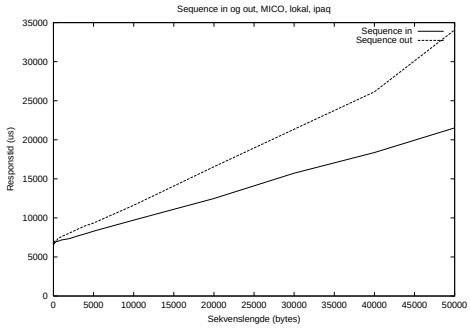
Tabell 6.15: Forhold mellom responstid for *sequence in* og *sequence out* mot MICO mellom pc1-ipaq, ipaq-pc1 og ipaq-ipaq2, beregnet med likning 6.5, side 96.



(a) pc1 (3)



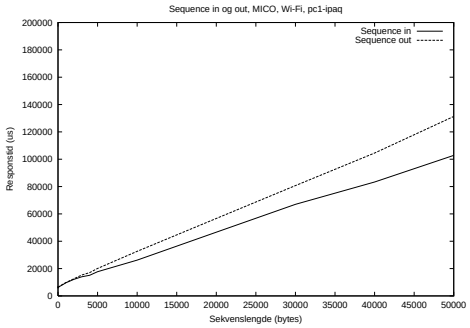
(b) pc2 (6)



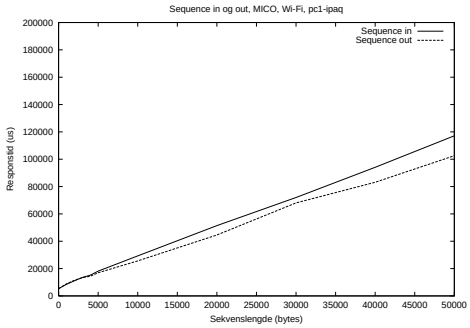
(c) ipaq (4)

Figur 6.8: *Sequence in* og *sequence out* kjørt mot MICO lokalt på ipaq, pc1 og pc2. **Merk:** Skalaene er ikke like for alle grafene.

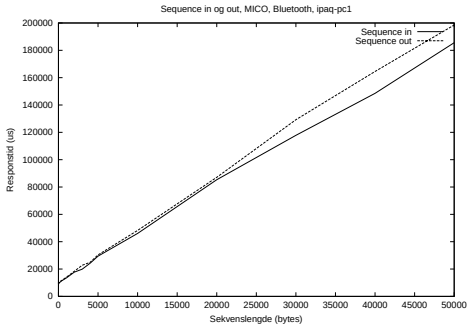
Som også metodekall-resultatene i forrige delkapittel viste er det klar forskjell mellom responstidene på de forskjellige maskinene. De to stasjonære maskinene har langt lavere responstid enn iPAQ-en, og pc1 er er en del raskere enn pc2. Den beste kombinasjonen av maskinvare er å la den raskeste maskinen stå for sending av data, ettersom dette er den tyngste operasjonen.



(a) pc1-ipaq (23)



(b) ipaq-pc1 (9)



(c) ipaq-ipaq2 (13)

Figur 6.9: *Sequence in* og *sequence out* kjørt mot MICO på pc1-ipaq, ipaq-ipaq2 og ipaq-pc1.

6.2.4 Dispatcher

Vi har sett på responstider med henholdsvis 1 og 20.000 objekter på tjenersiden for ulike maskinkonfigurasjoner. Dette har vi gjort både lokalt på enkeltmaskiner og mellom to maskiner i nettverk for å kunne se hvordan ulike maskiner på klient- og tjenersiden påvirker responstidene og skaleringsevne. Grunnen til at vi har presentert både Wi-Fi og Bluetooth, og ikke bare Wi-Fi som i forrige delkapittel, er at Bluetooth-målingene viser et annet forhold mellom maskinvarekonfigurasjonene enn Wi-Fi. Tabell 6.16 og 6.17 viser innsamlede data fra disse målingene.

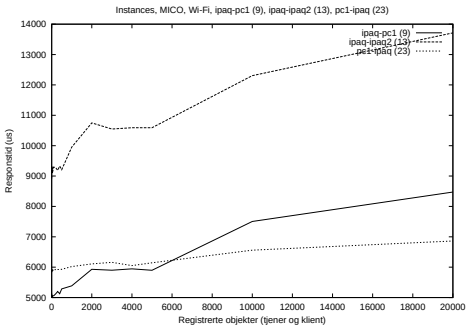
	Maskiner			*Responstid* 1 obj.	20.000 obj.	*Økning*	
lokal	pc1		(3)	739μs	839μs	100μs	14%
	ipaq		(4)	6.343μs	10.831μs	4.488μs	71%
Wi-Fi	pc1	ipaq	(23)	5.834μs	6.862μs	1.028μs	17%
	ipaq	pc1	(9)	5.036μs	8.473μs	3.437μs	68%
	ipaq	ipaq2	(13)	8.919μs	13.713μs	4.794μs	54%
BT	pc1	ipaq	(21)	59.764μs	60.251μs	487μs	1%
	ipaq	pc1	(5&11)	63.588μs	77.626μs	14.068μs	22%
	ipaq	ipaq2	(15&17&19)	79.150μs	90.489μs	11.339μs	14%

Tabell 6.16: Data fra *Instances* for MICO.

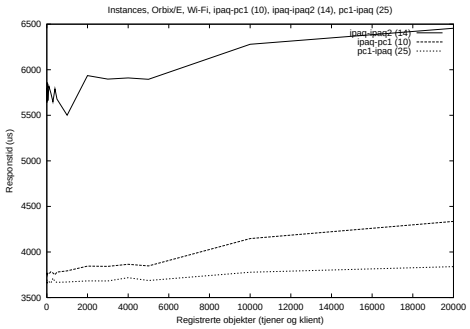
	Maskiner			*Responstid* 1 obj.	10.000 obj.	*Økning*	
lokal	pc1		(1)	246μs	252μs	6μs	2%
	ipaq		(4)	1.685μs	1.961μs	276μs	16%
Wi-Fi	pc1	ipaq	(25)	3.675μs	3.778μs	103μs	3%
	ipaq	pc1	(10)	3.756μs	4.147μs	391μs	10%
	ipaq	ipaq2	(14)	5.781μs	6.279μs	498μs	9%
BT	pc1	ipaq	(22)	55.500μs	55.529μs	29μs	$<<$1%
	ipaq	pc1	(2&12&29)	58.994μs	60.904μs	1.910μs	3%
	ipaq	ipaq2	(16&18&20)	75.896μs	77.238μs	1.342μs	2%

Tabell 6.17: Data fra *Instances* for Orbix/E. **Merk:** Maksimalt antall objekter her er 10.000.

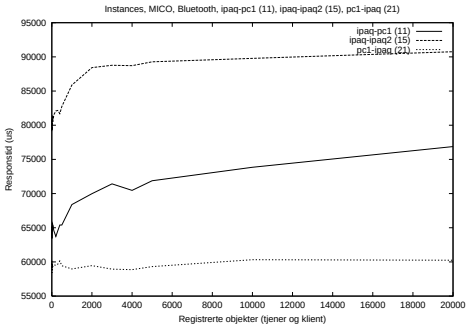
Data fra lokale målinger på arbeidstasjonen og iPAQ viser at responstiden



(a) MICO Wi-Fi (9, 13, 23)



(b) Orbix/E Wi-Fi (10, 14, 25)



(c) MICO Bluetooth (11, 15, 21)

Figur 6.10: *Instances* for ipaq-pc1, ipaq-ipaq2 og pc1-ipaq (*Merk:* Skalaene er ikke like for alle grafene.)

er betydelig lavere på arbeidsstasjonen. I tillegg ser vi at både den absolutte og prosentvise økningen i responstid fra 1 til 20.000 (10.000 for Orbix/E) objekter er svakere enn på iPAQ-en.

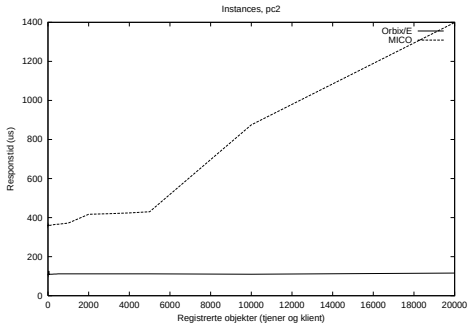
For forskjellige maskinkonfigurasjoner i nettverk ser vi at målinger mellom to iPAQ-er alltid gir høyere responstider enn når en arbeidsstasjon er involvert. Om man sammenlikner klient-tjener-konfigurasjonene *pc1-ipaq* og *ipaq-pc1* finner vi at for maksimalt antall objekter, 20.000 og 10.000 for henholdsvis MICO og Orbix/E, gir *pc1-ipaq* lavest responstid. For responstider med 1 objekt på tjenersiden har vi den samme fordelingen ved bruk av Bluetooth. Figur 6.10(c) viser dette.

Over Wi-Fi ser vi den samme tendensen for Orbix/E (se delfigur 6.10(b)). Under målinger med MICO finner vi imidlertid at vi oppnår best respons-

tid med klient-tjener-konfigurasjonen *ipaq-pc1*. Dette samsvarer med funnene fra metodekall-målinger i delkapittel 6.2.2. Figur 6.10(a) viser denne tendensen, og vi ser her at grafene krysser hverandre ved omtrent 6000 objekter (vi har ingen måledata mellom 5000 og 10000, og det er derfor vanskelig å si nøyaktig hvor grafene krysser).

For MICO over Wi-Fi ser det altså ut til at en arbeidsstasjon på tjenersiden gir best responstider opptil noen tusen objekter. Når antallet objekter blir høyere, vil en arbeidsstasjon som klient gi best resultat, da kombinasjonen *ipaq-pc1* skalerer dårligere. *Open CORBA Benchmarking* inkluderer metodekallene for å finne et tilfeldig objekt (se 4.1.4, side 53) i tidsmålingene, og dette vil påvirke målingene. Om denne rutinen tar lengre tid å utføre på iPAQ kan dette være årsaken til ulikhetene mellom maskinkonfigurasjonene.

Om vi tar for oss tallene for prosentvis økning i responstid, ser vi at vi i grove trekk har den samme fordelingen mellom maskin-konfigurasjonene: *pc1-ipaq* har minst økning, etterfulgt av *ipaq-ipaq2* og *ipaq-pc1*. De absolutte tallene for økning viser den samme fordelingen over Bluetooth. Med Wi-Fi fant vi derimot at *ipaq-pc1* øker mindre enn *ipaq-ipaq2*. Vi har ikke funnet noen forklaring på hvorfor nettverket påvirker skaleringen på denne måten.



Figur 6.11: *Instances* for Orbix/E (7) og MICO (6) på pc2.

Under lokale målinger på pc2 registrerte vi en drastisk økning i responstiden til MICO. Figur 6.11 viser denne tendensen. Mellom 5.000 og 10.000 objekter øker plutselig responstiden dramatisk. Vi har klart å reprodusere dette resultatet på pc2, men finner ingen liknende resultater i målingene fra de to andre testmaskinene. Heller ikke ved bruk av Orbix/E på pc2 skjer dette.

Da vi ikke så den samme tendensen på pc1, hadde vi en teori om at denne

økningen skyldtes maskinvare. pc2 har mindre minne (se 3.5, side 45) og andrenivå-cache enn pc1 (256 MB mot 512 MB), og vi sammenliknet derfor antall *page faults* under dispatcher-testen for pc2-målinger (6 og 24) med pc1-målingen (3). For pc2 ligger antall page faults betydelig høyere enn for pc1: måling 6 og 24 har henholdsvis 27235 og 28600, mens måling 3 har 4562. Vi logger tidspunktet for når *dispatcher*-målingen starter og slutter, men har dessverre ingen data på når *Open CORBA Benchmarking* utfører de forskjellige målingene for antall objekter innenfor dette tidsrommet. Det er derfor vanskelig med sikkerhet å avgjøre om økningen i page faults er sammenfallende med økningen vi ser i responstid — vi kan bare fastslå at det skjer under *dispatcher*-testen.

Vi ser altså at den maskinvarekombinasjonen som skalerer desidert best er *pc1-ipaq*. Vi antok at en kraftig maskin på tjenersiden ville føre til mer effektiv *dispatching*, men resultatene her tyder på vi oppnår bedre resultater skaleringsmessig når vi har den kraftigste maskinen på klientsiden. Vi har imidlertid funnet svakheter i OCBs målerutiner som kan være årsaken til dette, og nøler derfor med å trekke en bastant konklusjon basert på disse dataene.

Videre fant vi at kombinasjonen med to iPAQ-er skalerer dårligere enn *ipaq-pc1*-kombinasjonen under målinger over Wi-Fi, mens situasjonen er omvendt ved bruk av Bluetooth. Signifikansen av skaleringsevnen vises i tallene for prosentvis økning, hvor vi finner at selv om *ipaq-pc1*-kombinasjonen skalerer bedre enn *ipaq-ipaq2* over Wi-Fi, utgjør økningen i responstid for førstnevnte kombinasjon en større del av den totale responstiden, og har således større innvirkning.

6.2.5 Oppsummering

Som forventet viste målingene at iPAQ har langt lavere ytelse enn arbeidsstasjonene vi benyttet. Alle testene ga høyere responstid når bare iPAQ-er var involvert; både for *metodekall*-testene og *marshalling*-testene opplevde vi en kraftig reduksjon i responstiden når vi i stedet for to iPAQ-er brukte én arbeidsstasjon og iPAQ.

Måling av gjennomstrømning indikerer at det er mer krevende å sende data enn å motta dem, og maskinkonfigurasjonen fikk dermed merkbar innvirkning på responstiden ved økende størrelse på parameter- og returdata. I konfigurasjoner med arbeidsstasjon som klient og iPAQ-en som

tjener var responstidene lavest ved sending av data (både av parameter- og returverdier), mens situasjonen var omvendt om man byttet om klient-tjener-rollen til de to maskinene.

Lokale *dispatcher*-målinger viste at det er en betydelig forskjell i skalering på arbeidsstasjon og iPAQ. Sistnevnte har en absolutt og prosentvis økning som ligger langt over de vi ser for arbeidsstasjonen. For ulike maskinvarekombinasjoner fant vi at *pc1-ipaq* skalerer desidert best, både hva gjelder absolutt og prosentvis økning. Hvordan de to andre kombinasjonene, *ipaq-pc1* og *ipaq-ipaq2*, forholdt seg til hverandre berodde på typen nettverk som ble benyttet. Dette gjelder imidlertid ikke for prosentvis økning, hvor kombinasjonen med to iPAQ-er skalerte best uavhengig av nettverkstype. Vi har imidlertid mistanke om at dette skyldes en svakhet ved *Open CORBA Benchmarking*.

6.3 Nettverk

I dette delkapittelet tar vi for oss nettverkets innvirkning på ytelsen til CORBA-implementasjonene. Vi har sammenliknet resultater ved bruk av MICO og Orbix/E med forskjellige nettverkstyper og sett på hvordan forholdet mellom de to CORBA-implementasjonen har blitt påvirket av nettverket. Videre har vi studert forholdet mellom ytelsen til ulike maskinvare-konfigurasjoner og hvordan dette forholdet har endret seg ved bytte av nettverksteknologi.

6.3.1 Måling av gjennomstrømning og forsinkelse

Gjennomstrømning, her definert som den mengden av data (eller nyttelast) man får overført i et gitt tidsrom, er sjelden så høy som de teoretiske spesifikasjonene for båndbredde angir. I vår analyse vil vi studere gjennomstrømning og forsinkelse for ulike typer målinger i forskjellige testkonfigurasjoner. I den forbindelse er det interessant for oss å vite hvordan ytelsen er lokalt og på nettverksnivå — både for gjennomstrømning og forsinkelse. Disse resultatene kan så brukes som referanse under analyse av resultatene fra ytelsesmålingene.

For å samle data om nettverksytelse har vi gjort målinger over de aktuelle nettverksteknologiene mellom iPAQ-er, og mellom iPAQ og arbeidsstasjon, samt lokalt på disse maskinene. Til dette arbeidet har vi brukt det kommandolinje-baserte verkøyet `ttcp`. Ved å starte en prosess som mottar data på den ene siden av nettverksforbindelsen, og én som sender på den andre siden, har vi målt gjennomstrømning.

Dataene vi har overført er tilfeldig genererte data. I utgangspunktet hentet vi dataene fra en fil med av komprimerte data, men fant etter innledende målinger at resultatene ble påvirket av ytelsen til lagringsmediet, og valgte derfor å benytte `ttcp`s funksjonalitet for generering av data. Målingene har hatt en varighet på omtrent fem minutter, og er gjentatt flere ganger for hver konfigurasjon. Tabell 6.18 gir en oversikt over resultatet fra målingene.

Under måling av Bluetooth-forbindelsen gjorde vi målinger både med og uten komprimering i PPP-laget. *Open CORBA Benchmarking* genererer tilfeldige data under marshalling-målingene, og vi hadde derfor behov for

Nettverkstype	*Teoretisk båndbredde.*	*Målt ipaq - pc1*	*Målt ipaq - ipaq2*
802.11b	11.000 Kbps	3.600 Kbps	2.500 Kbps
Bluetooth	110-430 Kbps	80 Kbps	75 Kbps
Bluetooth (komp.)		2.800 Kbps	2.200 Kbps

Tabell 6.18: Målt gjennomstrøming mot teoretisk båndbredde (målt med `ttcp`).

Maskin	*Målt gjennomstrømning*
ipaq	127.000 Kbps
pc1	540.000 Kbps
pc2	1.320.000 Kbps

Tabell 6.19: Målt gjennomstrøming lokalt uten nettverk (målt med `ttcp`).

referansemålinger med komprimering. Målinger uten komprimering[3] ble gjort for å få en idé om hvordan den reelle gjennomstrømningen er i forhold til den teoretiske båndbredden.

Vi har også forsøkt å gjøre Bluetooth-målinger både med *5-slot*-pakker og *single slot*-pakker, uten at vi så noen merkbar ytelsesforskjell. Teoretisk sett skulle det være mulig å se forskjell på ytelsen. Det er derfor mulig at vi ikke har fått til å sette pakketypen riktig, og derfor kun har brukt én av typene under målinger. Under ytelsesmålingene med OCB har vi heller ikke kunnet finne nevneverdige forskjeller, og presentasjonen skiller derfor ikke mellom pakketypene.

Av resultatene i tabell 6.18 ser vi at den målte gjennomstrømningen er betydelig lavere enn den teoretiske båndbredden, både for 802.11b og Bluetooth. Dette er ikke spesielt oppsiktsvekkenede, da den oppgitte teoretiske båndbredden er hva nettverksteknologien maksimalt tillater, og inkluderer ikke overføring av *checksummer*, pakke-*headere* og andre kontrolldata som blir lagt til. De målte dataene viser kun *nyttelast*-data som sett fra et applikasjonssynspunkt. I [14], som omtaler denne situasjonen for Bluetooths vedkommende, blir det kommentert at oppgitte data er langt fra det man kan forvente seg på applikasjonsnivå. For symmetrisk overføring oppgir forfatterne tall fra 25 Kbit/s til 128 Kbit/s avhengig av pakketype, men presiserer at disse tallene er noe lave for eksempelets skyld. Vi ser av tabellen at våre måledata ligger innenfor dette området.

[3]Komprimering ble slått av i PPP-demonen med parametrene `nobsdcomp` og `nodeflate`.

Nettverkstype	*pc1 - ipaq*	*ipaq - pc1*	*ipaq - ipaq2*
802.11b (ukryptert)	2,7ms	2,7ms	3,8ms
Bluetooth	64ms	73ms	88ms
Bluetooth (komp.)	49ms	57ms	88ms

Tabell 6.20: RTT (i ms) mellom maskiner i testoppsettet for forskjellige nettverkstyper (målt med `ping`).

Maskin	*Målt RTT*
ipaq	0,2ms
pc1	0,03ms
pc2	0.04ms

Tabell 6.21: RTT lokalt på maskiner i testoppsettet (målt med `ping`).

På iPAQ H3870 kommuniserer Bluetooth-brikken med resten av iPAQ-en via et serielt grensesnitt. For Linux-installasjon anbefales det at man setter overføringshastigheten på brikken som tilbyr dette grensesnittet til 115.200kbps. Dette vil kunne begrense gjennomstrømningen. Ansatte ved Compaqs *Cambrigde Research Lab* har imidlertid funnet at denne også kan operere på 230.400kbps, men vi har ikke fått gjort målinger ved denne hastigheten.

For å ha en referanse til responstidene målt med *Open CORBA Benchmarking* har vi gjort enkle responstidmålinger med `ping` for å måle forsinkelsen. Resulatet er vist i tabell 6.20 og 6.21.

6.3.2 Metodekall

Vi har sett på hvordan ulike nettverksteknologier har påvirket responstidene. Ved å sammenlikne resultater fra målinger gjort over Wi-Fi og Bluetooth, samt lokale målinger, kan vi studere forholdet mellom responstider for ulike nettverkstyper. Vi har også sett på hvordan nettverket påvirker forholdet mellom responstid for de to CORBA-implementasjonene.

Tabell 6.22 viser et utvalg av responstider for målinger over forskjellige nettverkstyper med iPAQ-er. Resultatene viser oss at responstidene for Bluetooth-målinger ligger en størrelsesorden over tallene fra lokale og Wi-Fi-målinger. Transportprotokollen som benyttes, i vårt tilfelle TCP, har mest sannsynlig en innvirkning på måleresultatene ettersom protokollen benytter tre *segmenter* for oppkobling av forbindelsen og fire for

		Responstid			
	Nettverk	*Snitt*	*Q1*	*Median*	*Q3*
MICO	Lokal (4)	6.237	4.325	4.361	4.406
	Wi-Fi (13)	8.914	6.647	6.845	7.081
	Bluetooth (15)	80.550	73.739	79.476	82.458
Orbix/E	Lokal (31)	1.627	1.156	1.206	1.208
	Wi-Fi (14)	5.698	4.071	4.265	4.474
	Bluetooth (16)	76.050	73.666	74.970	81.288

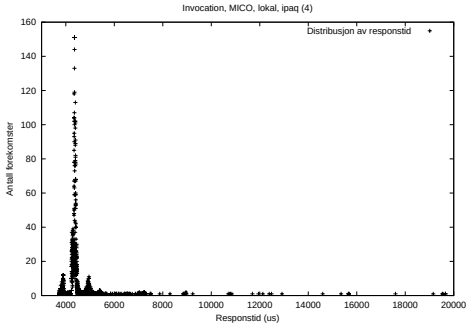
Tabell 6.22: Responstider for MICO og Orbix/E mellom iPAQ-er i forskjellige nettverk.

nedkobling[37]. Med andre ord sendes det syv meldinger over nettverket som *overhead* for forbindelsen, og tidsforbruket er naturlig nok avhengig av latensen til nettverksteknologien. Forskjellen mellom Wi-Fi og lokale målinger uten nettverk er ikke like stor som mellom Wi-Fi og Bluetooth, men vi ser her forskjeller mellom de to CORBA-implementasjonene. For MICO er responstidene under lokale målinger en god del høyere enn tilsvarende tall for Orbix/E. Vi har faktisk lavere responstider over Wi-Fi med Orbix/E enn lokalt med MICO.

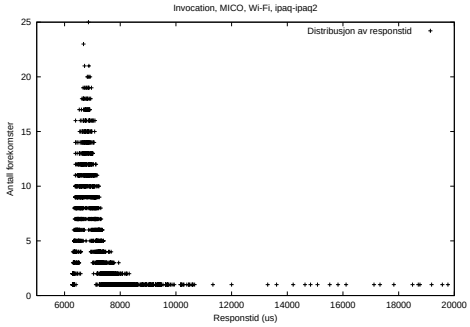
Figur 6.12 og 6.13 viser grafiske plot av responstiden til MICO og Orbix/E for de ulike nettverkstypene. I disse plottene ser vi en tendens i fordelingen av responstid, som ikke er like lett å få øye på ved å studere tallmaterialet i tabell 6.22. For lokale målinger og spesielt over Bluetooth ser vi at vi to eller flere større ansamlinger. Over Bluetooth er det to områder hvor de fleste forekomstene befinner seg, og snittet ligger for MICO midt i mellom disse toppene, mens det for Orbix/E ligger nærmere den venstre. For lokale målinger er tendensen flere småtopper spredt rundt en mere markant topp, som ligger omtrent på median-verdien.

$$forholdstall_{6.7} = \frac{rt_{Wi-Fi}}{rt_{Bluetooth}} \tag{6.7}$$

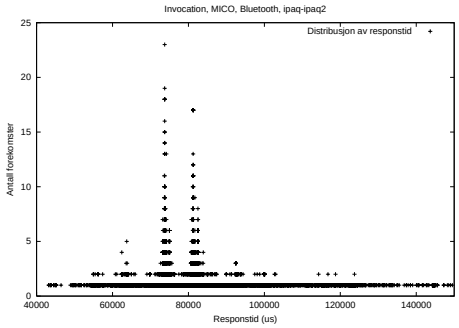
Tabell 6.23 viser forholdet, beregnet ved bruk av likning 6.7, mellom responstider med Wi-Fi og Bluetooth for Orbix/E og MICO ved forskjellige maskinvarekonfigurasjoner. For Orbix/E er dette tallet gjennomgående lavere enn for MICO. Dette betyr at å bytte nettverksteknologi har større innvirkning på responstidene for Orbix/E enn for MICO.



(a) Lokal (4)



(b) Wi-Fi (13)

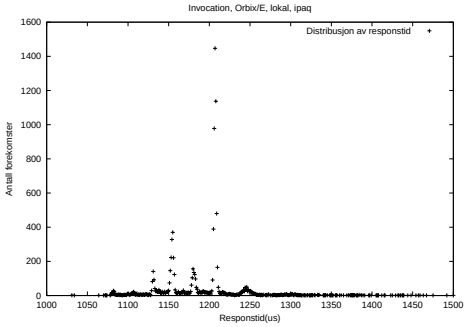


(c) Bluetooth (15)

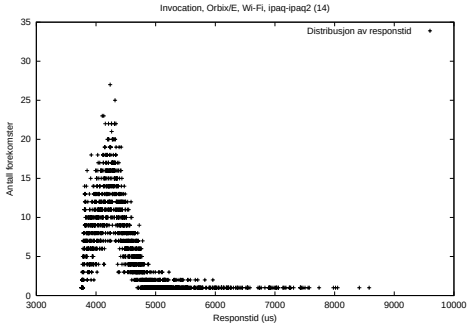
Figur 6.12: *Invocation* for MICO med forskjellige nettverkstyper.
Merk: Skalaene er ikke like for alle grafene.

Differansen i gjennomsnittlig responstid mellom Orbix/E og MICO varierer kraftig mellom lokale målinger og målinger over nettverk. (se tabell 6.24); ved bruk av nettverk er differansen en størrelsesorden høyere. Imidlertid ser man av forholdstallene at differansen utgjør en mindre del av den totale responstiden dess tregere nettverk man benytter. Forskjeller mellom ORB-er viskes altså ut ved tregt nettverk.

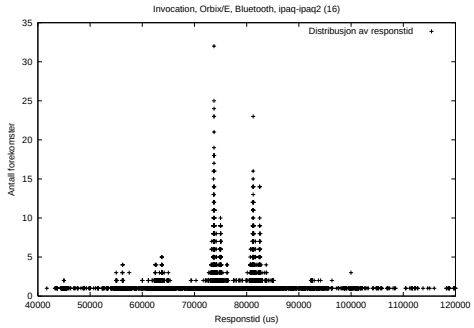
For responstider ved bruk av forskjellige nettverkstyper fant vi at Bluetooth er betydelig tregere enn Wi-Fi og målinger uten nettverk. For Orbix/E ser vi større forskjell mellom lokale målinger og Wi-Fi enn for MICO. Videre så vi at bytte av nettverksteknologi fra Wi-Fi til Bluetooth har størst innvirkning på responstidene ved bruk av Orbix/E. I tillegg viste målingene at forskjellen mellom CORBA-implementasjonene ble for-



(a) Lokal (31)



(b) Wi-Fi (14)



(c) Bluetooth (16)

Figur 6.13: *Invocation* for Orbix/E med forskjellige nettverkstyper.
Merk: Skalaene er ikke like for alle grafene.

svinnende liten ved tregere nettverk, da de gjennomsnittlige responstidene da er mye høyere.

	Klient	*Tjener*	$Forholdstall_{6.7}$
MICO	ipaq	ipaq2	0,11
	ipaq	pc1	0,08
	pc1	ipaq	0,10
Orbix/E	ipaq	ipaq2	0,07
	ipaq	pc1	0,06
	pc1	ipaq	0,06

Tabell 6.23: Sammenlikning av forholdstall for responstid mellom Wi-Fi og Bluetooth.

	$Forholdstall_{6.1}$		
Nettverk	*Snitt*	*Median*	*Diff. for snitt*
Lokal	0,3	0,3	2-500
Wi-Fi	0,6	0,6	2-3.000
Bluetooth	0,9-1,0	0,9-1,0	2-7.000

Tabell 6.24: Sammenlikning av forholdstall og differanse i responstid mellom Orbix/E og MICO for ulike typer nettverk (likning 6.1 er brukt til utregning av forholdstallet).

	Test	Nettverk		0 KB	50 KB	Økning	
MICO	In	lokal	(4)	6.650μs	21.522μs	14.872μs	224%
		Wi-Fi	(13)	9.494μs	185.571μs	176.077μs	1.855%
		Bluetooth	(15)	81.011μs	354.318μs	273.307μs	338%
	Out	lokal	(4)	6.827μs	34.070μs	27.243μs	399%
		Wi-Fi	(13)	9.454μs	198.528μs	189.074μs	2.000%
		Bluetooth	(15)	82.389μs	368.277μs	285.888μs	347%
Orbix/E	In	lokal	(31)	1.648μs	6.917μs	5.269μs	320%
		Wi-Fi	(14)	5.688μs	170.995μs	165.307μs	2.906%
		Bluetooth	(16)	76.018μs	358.839μs	282.821μs	372%
	Out	lokal	(31)	1.872μs	9.551μs	7.679μs	410%
		Wi-Fi	(14)	5.870μs	171.819μs	165.949μs	2.827%
		Bluetooth	(16)	76.218μs	369.670μs	293.452μs	385%

Tabell 6.25: Data fra *sequence in/out* mot Orbix/E og MICO på iPAQ.

6.3.3 Marshalling

For å studere forskjellene i gjennomstrømning med forskjellige nettverk, har vi sett nærmere på målinger gjort med begge ORB-ene lokalt på iPAQ og mellom to iPAQ-er med Bluetooth- og Wi-Fi-nettverk. Tabell 6.25 inneholder resultatene fra målinger gjort uten dataoverføring og med 50 KB datamengde. I tillegg er de absolutte og relative differansene mellom de to målingene beregnet.

Målingene gjort over Wi-Fi har en relativt lav responstid ved rene metodekall (ping roundtrip), i forhold til lokale målinger (se figur 6.14 og 6.15). Bluetooth-målingene har derimot responstider som ligger rundt ti ganger høyere enn Wi-Fi, noe som tyder på at oppkoblingen av TCP-forbindelsen tar betydelig lengre tid over Bluetooth enn Wi-Fi (se forøvrig delkapittel 6.3.2). Ved dataoverføring på 50 KB er den relative forskjellen langt mindre; Wi-Fi-responstidene utgjør rundt 2/3 av responstidene for Bluetooth-målingene mot MICO. For Orbix/E er forskjellen noe større. Responstidene for lokale målinger ligger langt under nettverksmålingene, og utgjør kun 5-10% av nettverks-responstidene.

Sett i forhold til den store forskjellen i den teoretiske båndbredden de to nettverksteknologiene tilbyr, henholdsvis 11 Mbps for Wi-Fi og i underkant av 0,5 Mbps for Bluetooth, virker våre resultater noe unormale. En del av forklaringen ligger i at den faktiske gjennomstrømningen er en god del lavere enn den oppgitte båndbredden. I tillegg har vi benyttet komprime-

		$Forholdstall_{6.8}$	
	Nettverk	*0 KB*	*50 KB*
MICO	lokal (4)	0,97	0,63
	Wi-Fi (13)	0,98	0,93
	Bluetooth (15)	1,00	0,96
Orbix/E	lokal (31)	0,84	0,72
	Wi-Fi (14)	0,97	1,00
	Bluetooth (16)	1,01	0,97

Tabell 6.26: Forholdstall mellom responstid for *sequence in* og *sequence out* mot MICO og Orbix/E lokalt på ipaq samt mellom ipaq-ipaq2 med Wi-Fi og Bluetooth, beregnet med likning 6.8.

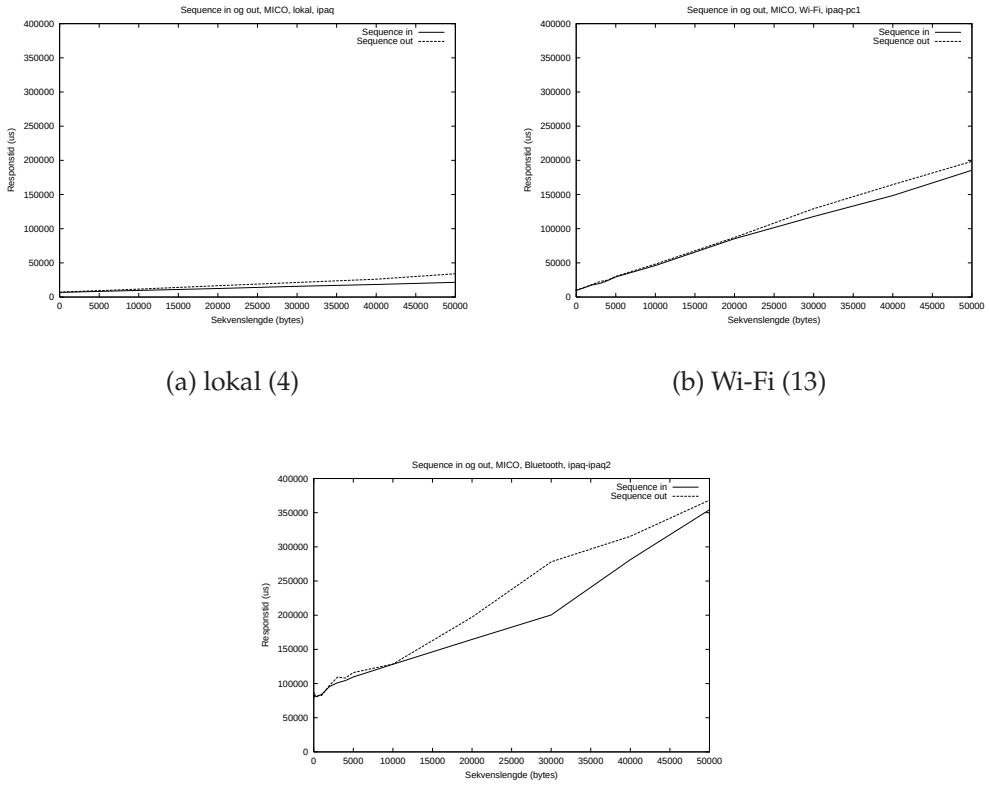
ring på PPP-forbindelsen, noe som medfører at gjennomstrømningen over Bluetooth-nettverket blir langt nærmere gjennomstrømningen for Wi-Fi. Se delkapittel 6.3.1, side 106, for mer informasjon om våre målinger av gjennomstrømningen til de forskjellige nettverkstypene.

Målingene gjort over Bluetooth med forskjellige pakkestørrelser (*single slot* og *5-slot*, se delkapittel 2.5.2, side 20, viste ingen merkbare forskjeller i responstiden ved overføring av større datamengder. Nettopp ved marshalling-testene ventet vi å se en differanse mellom målingene med forskjellige pakkestørrelser, og mangelen på forskjeller støtter opp under mistanken om at vi ikke har klart å konfigurere Bluetooth-forbindelsen til å bruke *5-slot*-pakker.

Som vi tidligere har påpekt er det en markant forskjell mellom responstiden for *sequence in*- og *sequence out*-testene. Forholdstallet mellom måleresultatene fra disse testene er presentert i tabell 6.26, beregnet med likning 6.8. Ved lokale målinger er det en markant, relativ forskjell mellom responstidene for *sequence in* og *out* med 50 KB dataoverføring. Med Wi-Fi er forskjellen enda mindre og for Orbix/E er responstidene tilnærmet identiske. Med Bluetooth er også den relative forskjellen mellom *sequence in* og *out* svært liten. Forsinkelsen fra de trådløse nettverkene overskygger altså forskjellen mellom de to sekvenstestene.

$$forholdstall_{6.8} = \frac{responstid_{out}}{responstid_{in}} \tag{6.8}$$

Ved lokale målinger har vi naturlig nok kjørt klient og tjener på samme

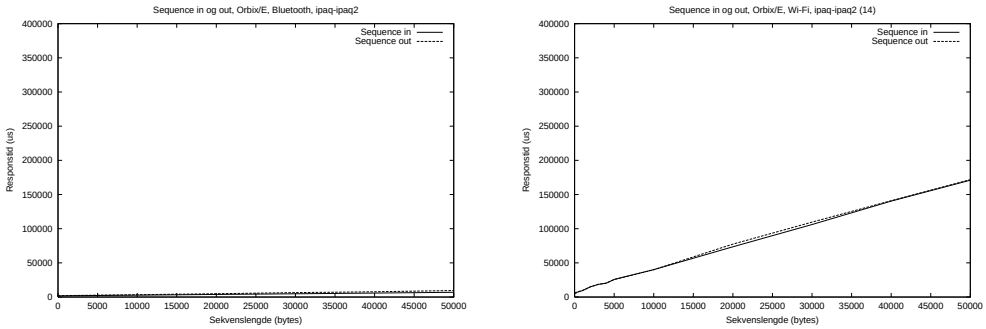


(a) lokal (4)

(b) Wi-Fi (13)

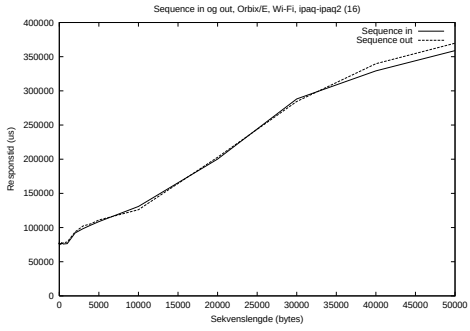
(c) Bluetooth (15)

Figur 6.14: *Sequence in* og *sequence out* kjørt mot MICO på ipaq.



(a) lokal (31)

(b) Wi-Fi (14)



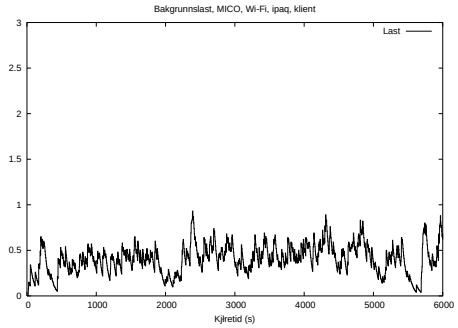
(c) Bluetooth (16)

Figur 6.15: *Sequence in* og *sequence out* kjørt mot Orbix/E på ipaq.

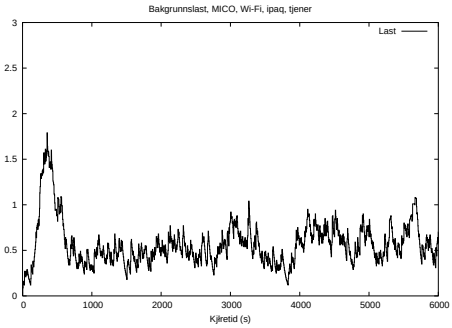
maskin, mens vi har benyttet to separate maskiner til nettverksmålingene. Maskinen for den lokale målingen hadde derfor høyere last enn hver av de to maskinene i nettverksmålingene. Figur 6.16 viser bakgrunnslasten målt på maskinene under ytelsesmålingene. Dette er en svakhet i testoppsettet og vi antar at responstidene for de lokale målingene er noe høyere enn de ville vært om ikke maskinen hadde hatt så høy bakgrunnslast.

Målingene våre viser at de trådløse nettverkene innfører en betydelig forsinkelse ved overføring av større datamengder. Gjennomstrømningen over Wi-Fi og Bluetooth er langt mindre enn for de lokale målingene, mens forskjellen mellom gjennomstrømningen til Bluetooth og Wi-Fi er relativt liten i forhold til nettverkenes tekniske spesifikasjoner. Dette skyldtes at dataene komprimeres før de sendes over Bluetooth-nettet, mens dataene sendes ukomprimert over Wi-Fi.

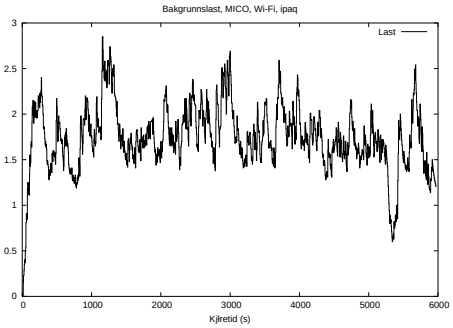
Uten dataoverføring er det hovedsakelig oppkobling av nettverksforbindelse som tar tid, og her er det en betydelig forskjell mellom Wi-Fi og Bluetooth; Bluetooth har langt høyere latens enn Wi-Fi. Forsinkelsene som bruk av Wi-Fi og Bluetooth medfører gjør at den relative forskjellen mellom sekvenstestene og ORB-ene blir langt mindre enn ved metodekall uten dataoverføring.



(a) Klient, ipaq (13)



(b) Tjener, ipaq2 (13)



(c) Klient og tjener, ipaq (4)

Figur 6.16: Bakgrunnslast på klient og tjener under måling gjort mot MICO på iPAQ.

6.3.4 Dispatcher

For å studere innvirkningen nettverket har på skaleringsevnen til CORBA-implementasjonene utførte vi målinger over Wi-Fi og Bluetooth. Dette ble gjort for både Orbix/E og MICO, og et utdrag av resultatene fra disse målingene, i tillegg til data fra lokale målinger, vises i tabell 6.27.

		Responstid			
	Nettverk	*1 obj.*	*20.000 obj.*	*Økning*	
MICO	lokal (4)	6.343μs	10.831μs	4.488μs	71%
	Wi-Fi (13)	8.919μs	13.713μs	4.794μs	54%
	Bluetooth (15)	79.134μs	90.750μs	11.616μs	15%
Orbix/E	lokal (31)	1.685μs	1.961μs	276μs	16%
	Wi-Fi (14)	5.781μs	6.455μs	674μs	12%
	Bluetooth (16)	75.972μs	78.419μs	2.447μs	3%

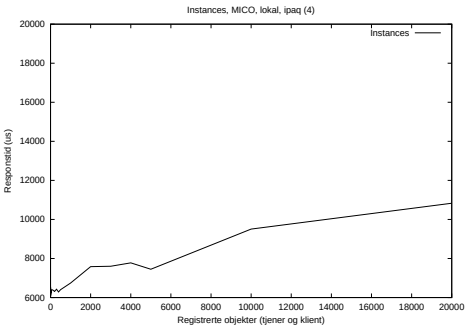
Tabell 6.27: Data fra *Instances* mellom iPAQ-er for forskjellige nettverk (merk at maks antall objekter for måling 31 kun er 10.000).

Av tabellen kan vi se at økningen i responstid er større for Bluetooth enn Wi-Fi. I tillegg finner vi at økningen ved lokale målinger uten nettverk er lavere enn ved bruk av nettverk. Dette gjelder for både Orbix/E og MICO. Det ser altså ut til at ved økning i antall objekter på tjenersiden skalerer CORBA-implementasjonene dårligere jo tregere nettverk som brukes.

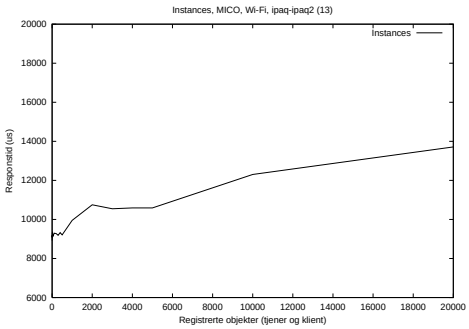
Imidlertid stiger også den gjennomsnittlige responstiden betraktelig ved tregere nettverk. Vi ser av tallene for prosentvis økning i responstid i tabell 6.27 at endringen i responstid, på tross av at den øker, har mindre innvirkning når nettet blir tregere. I praksis vil dette si at forskjellen i skaleringsevne mellom CORBA-implementasjonene blir forholdsvis liten når man bruker trege nettverk.

	Responstid			
ORB	*1 obj.*	*2.000 obj.*	*Økning*	
MICO (15)	79.134μs	88.424μs	9.290μs	12%
Orbix/E (16)	75.972μs	77.422μs	1.450μs	2%

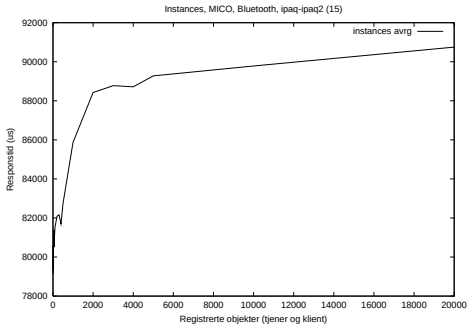
Tabell 6.28: Data fra *Instances* (1 til 2.000 objekter) mellom iPAQ-er for Bluetooth.



(a) lokal (4)



(b) Wi-Fi (13)



(c) Bluetooth (15)

Figur 6.17: *Instances* kjørt mot MICO på ipaq.
Merk: Skalaene er ikke like for alle grafene.

Ved bruk av Bluetooth opplever vi de største forskjellene i skaleringsevne i området 1 til 2.000 objekter. MICO har i dette området, for målinger mellom iPAQ-er, en økning i responstid som utgjør en betydelig del av den totale økningen. Av tabell 6.28 kan vi se at vi for MICO utgjør økningen i responstid fra 1 til 2.000 objekter 12 prosentpoeng av 15 totalt. I figur 6.17(c) vises denne tendensen tydelig. For Orbix/E er tilsvarende tall 2 av 3 prosentpoeng. Ser vi på de absolutte tallene for økning er ikke forskjellen så dramatisk; her utgjør MICOs økning for de første 2.000 objektene 80% av den totale økningen, mens tallet for Orbix/E er 60%.

Målingene viser at dess tregere nettverksteknologi som benyttes, dess dårligere skalerer responstidene ved økende antall objekter. Imidlertid øker også den gjennomsnittlige responstiden betraktelig ved bruk av tregt nett-

verk, og signifikansen av denne svekkelsen i skaleringsevne blir derfor mindre, men vi har ikke funnet noen forklaring på dette.

Vi så også i delkapittel 6.2.4 at nettverksvalget fikk innvirkning på fordelingen i skaleringsevne mellom de forskjellige maskinvarekonfigurasjonene. Hvilken klient-tjener-kombinasjon av pc1 og ipaq som ga best skaleringsresultat var avhengig om Wi-Fi eller Bluetooth ble brukt.

6.3.5 Oppsummering

Målinger av metodekall viste at bruk av Bluetooth gir betraktelig høyere responstider enn ved Wi-Fi og lokale målinger uten nettverk. Den forsinkelsen som nettverket innførte, da særlig med Bluetooth, gjorde at forholdet mellom responstiden for de to CORBA-implementasjonene ble jevnere. Videre fant vi at bytte av nettverk påvirket responstidene i større grad ved bruk av Orbix/E enn ved MICO, og at forholdet mellom CORBA-implementasjonene ble forsvinnende liten ved tregere nettverk.

Uten overføring av parameter- eller returdata under marshalling-testene er forskjellen i responstid mellom lokale-, Wi-Fi- og Bluetooth-målinger store. Særlig verdiene for Bluetooth ligger høyt over de andre, noe vi også så under metodekall-målingene. Når vi øker størrelsen på overførte data minsker denne forskjellen. I særlig grad ser vi at forskjellen mellom Bluetooth og lokale samt Wi-Fi minsker. Dette kan i stor grad tilskrives komprimeringen. Videre ser vi at den relative forskjellen mellom responstider for sending av parameter- og returdata blir større ved tregere nettverk.

Også under dispatcher-målingene så vi at nettverket påvirker responstidene. Ved bruk av tregere nettverksteknologi skalerer responstidene ved økende antall objekter dårligere. Imidlertid er også latensen i nettverket kraftig, og signifikansen av forsinkelsen ved dispatching minsker derfor.

6.4 Forbruk av systemressurser

Under kjøring av ytelsesmålingene, logget vi ressursforbruket på maskinen (se delkapittel 5.2, side 59, for flere detaljer) for å kartlegge eventuelle flaskehalser. Spesielt viktig var det å få studert hvilke begrensninger maskinvaren til iPAQ-en eventuelt setter. Enkelte av loggene var mangelfulle for Orbix/E-målingene, og vi har hovedsakelig studert dataene fra MICO-målingene. Der Orbix/E-loggene var komplette viste dataene seg å være relativt like som for MICO.

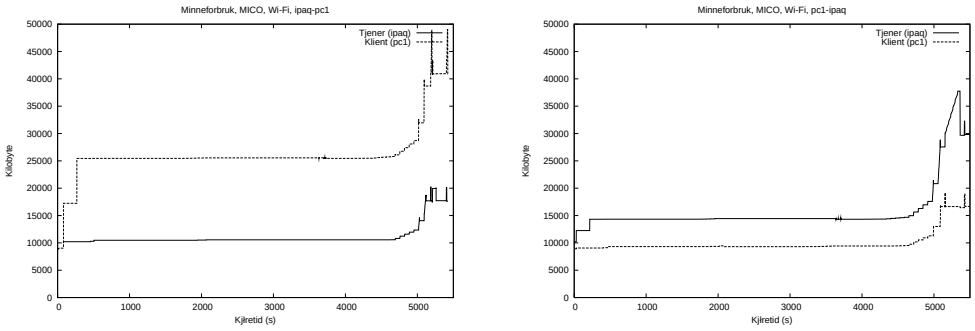
6.4.1 Minne

Klienten legger beslag på rundt 10 MB minne under kjøring av ytelsesmålingene, mens tjeneren bruker noe mer, avhengig av hvor mye minne som er tilgjengelig. Mot slutten av av ytelsesmålingen foretas skaleringstesten, som bruker langt mer minne enn de andre testene, noe som la begrensninger for hvor mange objekter som kunne brukes på iPAQ. Figur 6.18 viser en grafisk fremstilling av minneforbruket for to målinger, gjort med ipaq og pc1. Vi ser at klienten har brukt omtrentlig like mye minne på begge maskinene, mens tjeneren har brukt mer minne på pc1. Målingene av minneforbruket under Orbix/E-målingene, gir langt høyere verdier. På iPAQ går minneforbruket opp til rundt 250 MB under skaleringstesten, noe som er langt mer minne enn iPAQ har til rådighet. Vi har ikke funnet noen forklaring på hvorfor Linux-kjernen rapporterer at prosessene bruker rundt fire ganger mer minne enn maskinen er utstyrt med, og har derfor valgt å forkaste disse dataene.

Av loggene ser vi også at det er en betraktelig forskjell på antall *page faults* som tiltreffer under *dispatcher*-testen ved måling av Orbix/E og MICO. Linux holder oversikt over to typer *page faults*[4]: én som resulterer i en I/O-operasjon (lesing fra disk), *major page fault*, og én som håndteres uten I/O (for eksempel *Copy On Write*), *minor page fault*[38][35]. Under målingene for MICO forekom det rundt 1.000-1.500 *major pagefaults* og 70-100.000 *minor page faults*. For Orbix/E var derimot tallet langt lavere, med kun 2-500 *minor* og rundt 400 *major page faults*.

Mest sannsynlig skyldes forskjellen at MICO bruker mer minne enn Orbix/E. På en maskin med lite minne vil dette medføre flere *page faults*.

[4] Se `man proc`



(a) ipaq-pc1 (9) (b) pc1-ipaq (23)

Figur 6.18: Minneforbruk under måling gjort mellom ipaq og pc1 (9 og 23) mot MICO.

Dette resulterer i ekstra *overhead*[35] for prosessen, og er blant annet en av årsakene til at MICO yter dårligere enn Orbix/E.

6.4.2 Prosessor

Loggingen for MICO-målingens CPU-forbruk, viser oss at forbruket kun har vært 100% ved lokale målinger. Klienten og tjeneren har da hver brukt mellom 40% og 60%. For målingene mellom to maskiner, med Wi-Fi- eller Bluetooth-nett er ressursforbruket langt lavere. Dette skyldes at mye av tiden går med til *I/O-wait*, grunnet forsinkelsen i nettverket. Figur 6.19 viser CPU-forbruket for målinger gjort på iPAQ under en MICO-måling. Målingene som ble gjort lokalt og via Wi-Fi avsluttet tidligere enn Bluetooth-målingene, noe som skyldes at OCB avslutter hver deltest etter at et gitt antall målinger er foretatt. Under lokale målinger samt ved bruk av Wi-Fi, utføres testene raskere og målingene blir tidligere ferdige enn når Bluetooth-nett benyttes.

Loggene for Orbix/E er ikke komplette og data for tjeneren mangler fullstendig for alle målingene. Problemene med Orbix/E-målingene er beskrevet nærmere i delkapittel 5.5, side 76.

6.4.3 Nettverk

Loggene for nettverksgrensesnittene viser oss at nettverket kun belastes fullt ut i to av OCBs deltester: marshalling og senere i de kombinerte testene. Det var ikke nevneverdig forskjell mellom målingene for Orbix/E og MICO, men naturlig nok var valget av nettverksteknologi avgjørende for dataraten som ble brukt over nettverket. Figur 6.20 viser grafer av gjennomstrømningen for målinger gjort på to iPAQ-er, knyttet sammen med Wi-Fi- og Bluetooth-nett. Som beskrevet i foregående avsnitt er det forskjell i kjøretiden for målingene og dette er grunnen til at toppene i nettverksforbruket ikke er i samme tidspunkt for de to målingene. Under disse ytelsesmålingene ble gjennomstrømningene på det høyeste målt til oppunder 2.000 Kbps for Bluetooth og rundt 3.500 for Wi-Fi. Målingen av gjennomstrømningen er gjort mot PPP[5]-grensesnittet, som inkluderer komprimering, og verdiene stemmer da godt overens med målingen av gjennomstrømningen vi har presentert i delkapittel 6.3.1, side 106.

6.4.4 Oppsummering

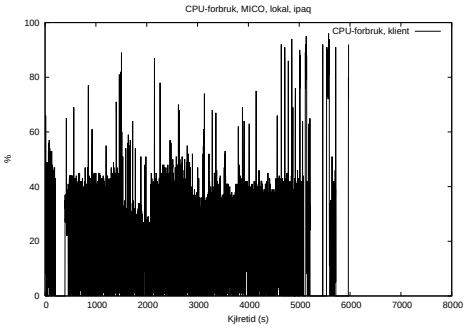
Minneforbruket under OCB-målingene varierer for tjener, klient og hvor mye minne maskinen er utstyrt ved. Kun under *dispatcher*-testen, hvor det opprettes et stort antall objekter, er minneforbruket ekstremt høyt. Den begrensede minnekapasiteten til iPAQ gjorde at *dispatcher*-testen måtte begrenses i omfang. Under ytelsesmålingene er det markant forskjell mellom Orbix/E og MICO når det gjelder hvor mange *page faults* som inntreffer — MICO har langt flere *page faults* under *dispatcher*-testen, og dette er en medvirkende årsak til at denne ORB-en har dårligere ytelse enn Orbix/E.

Forbruket av CPU-tid er knyttet til hvilken nettverksteknologi som benyttes. Ved bruk av Bluetooth er forbruket relativt lavt, grunnet mye *I/O-wait*, mens forbruket er en del høyere ved bruk av Wi-Fi. Under lokale ytelsesmålinger blir prosessoren belastet fullt ut og er i stor grad flaskehalsen i systemet.

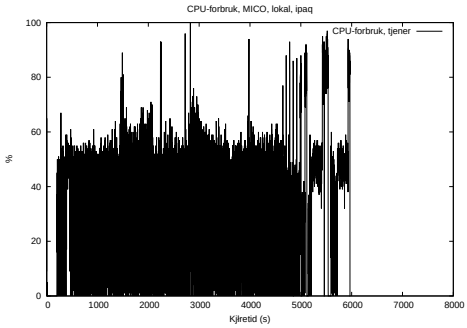
Den målte gjennomstrømningen er høyest for *marshalling* og de kombinerte testene. Verdiene ligger da oppunder resultatet fra gjennomstrømningsmålingene vi har gjort med `ttcp`, noe som betyr at nettverket har blitt belastet maksimalt under disse testene. Det er ikke noen vesentlige for-

[5]Point to Point Protocol

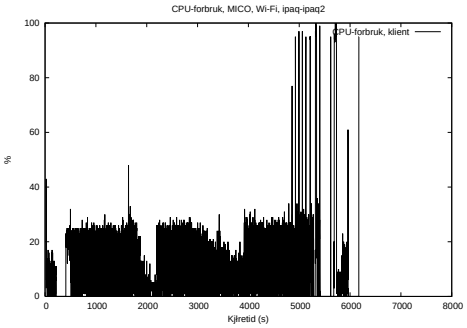
skjeller mellom resultatene fra Orbix/E- og MICO-testene, noe som også var forventet ettersom ORB-ene benytter seg av samme standard (IIOP) for å pakke inn data som skal sendes mellom CORBA-objektene.



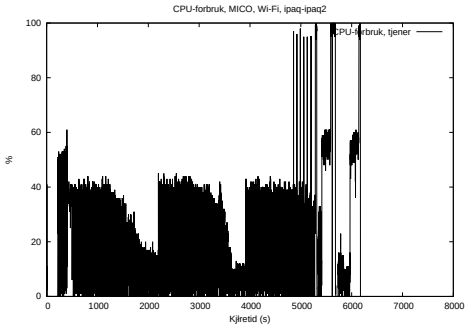
(a) Klient på ipaq, lokal (4)



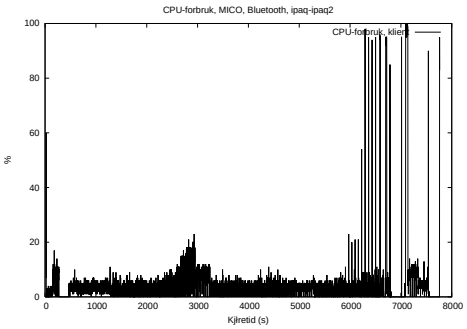
(b) Tjener på ipaq, lokal (4)



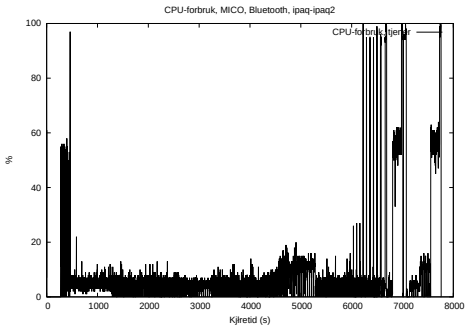
(c) Klient på ipaq, Wi-Fi (13)



(d) Tjener på pc1, Wi-Fi (13)

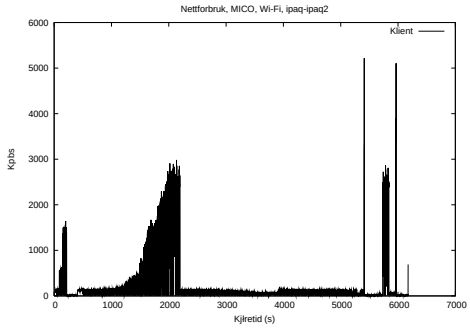


(e) Klient på ipaq, Bluetooth (15)

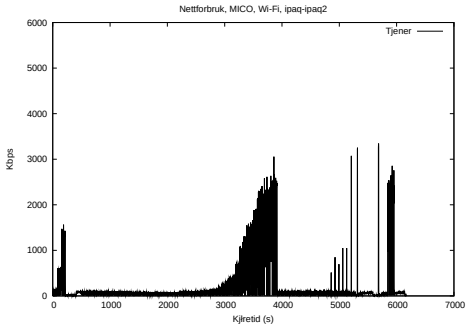


(f) Tjener på pc1, Bluetooth (15)

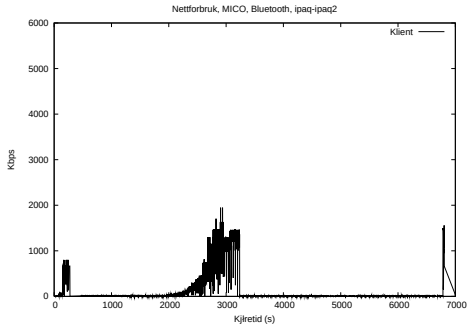
Figur 6.19: CPU-forbruket under målinger gjort på ipaq lokalt og med trådløse nett (MICO).



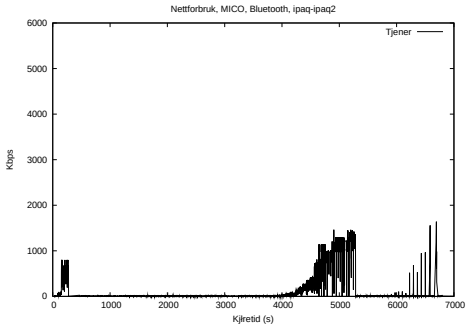
(a) Klient på ipaq, Wi-Fi (13)



(b) Tjener på ipaq2, Wi-Fi (13)



(c) Klient på ipaq, Bluetooth (15)



(d) Tjener på ipaq2, Bluetooth (15)

Figur 6.20: Forbruk av nettverk under målinger gjort på ipaq-ipaq2 over Wi-Fi (13) og Bluetooth (15).

6.5 Sammenlikning med andres resultater

Som nevnt i delkapittel 2.6 har vi ikke funnet annet forskningsarbeid som har fokusert på CORBA-implementasjoner kjørt på håndholdte enheter.

Douglas C. Schmidt og Aniruddha S. Gokhale har gjort ytelsesmålinger mot blant annet Orbix og VisiBroker [19][9]. Disse målingene ble imidlertid gjort i 1997, og Orbix er siden den gang omskrevet fullstendig og i tillegg tilpasset ressurssvake enheter. VisiBroker har vi ikke sett noe nærmere på, men det er ikke unaturlig å tro at også den har blitt forbedret i løpet av de siste fem årene. Skaleringsmålingene Schmidt og Gokhale foretok er dessuten ganske begrenset omfangsmessig i forhold til *Open CORBA Benchmarking*s tester. Blant annet var hverken VisiBroker eller Orbix dimensjonert for mange objekter, og begge ORB-ene krasjet ved oppretting av rundt 1.000 objekter. Til sammenlikning har vi med OCB kjørt ytelsestester med opptil 20.000 objekter på tjenersiden.

Resultatene fra arbeidet til Böszörményi, Wickner og Wolf, viser at mellomvare ikke nødvendigvis medfører stor *overhead*, og en optimalisert ORB (TAO) har relativt lave responstider. Våre resultater viser også at det er store forskjeller mellom CORBA-implementasjoner, men at forskjellen overskygges ved bruk av trege, trådløse nettverk.

Målsetning til *Open CORBA Benchmarking*-prosjektet er å lage et analyseverktøy som kan brukes til sammenlikning av ytelsen til forskjellige CORBA-implementasjoner (se forøvrig delkapittel 4.1, side 49). I den forbindelse har utviklerene opprettet en database der man kan legge inn resultater fra ytelsesmålinger. Både utviklerne av OCB og andre har bidratt med data fra målinger, men dessverre ser det ikke ut til at noen andre har gjort målinger på Intels StrongARM-prosessorer. Med andre ord har vi ingen sammenlikningsdata for våre målinger gjort på iPAQ. Databasen inneholder heller ikke informasjon om hvilken nettverksteknologi som er benyttet under målingene og vi har derfor ikke hatt mulighet til å søke etter målinger gjort over trådløse nettverk.

Ved søk i databasen fant vi kun ett sett med målinger for Orbix/E og MICO, gjort på samme maskin. Maskinen var noe raskere enn våre (900 MHz) og Orbix/E var benyttet i en noe eldre versjon (2.0). Resultatene fra disse målingene er relativt like våre resultater: Orbix/E har langt lavere responstid enn MICO.

6.6 Sammendrag

Vi har i dette kapittelet analysert data fra en rekke ytelsesmålinger gjort med testverktøyet *Open CORBA Benchmarking*. Under analysen har vi bruk tre forskjellige innfallsvinkler for å studere de innsamlede dataene. Vi har først tatt for oss målte forskjeller mellom de to CORBA-implementasjonene. Deretter har vi sett på ulik maskinvares innvirkning på resultatene, for så til slutt å studere to forskjellige trådløse nettverksteknologier. Vi har også sett på systemforbruk under målingene. Dette sammendraget gir en oversikt over det vi anser som de viktigste funnene fra analysen, samt en oppsummering av åpne spørsmål tilknyttet analyseresultatet — spørsmål vi dessverre ikke fikk anledning til å se nærmere på.

Under første del, hvor vi så på Orbix/E og MICO, fant vi en klar forskjell i responstid ved enkle metodekall. MICO har en responstid som er tre ganger så høy som Orbix/Es. Det viste seg også at Orbix/E ga bedre ytelse ved de andre målingene. Både under marshalling- og dispatchertestene ga denne CORBA-implementasjonen best resultater med hensyn på skaleringsevne, noe som indikerer at Orbix/E har en mer effektiv *dispatcher* og en IDL-kompilator som genererer raskere marshallingrutiner. Marshalling-målinger viste videre en forskjell mellom sending og mottak av data. Ved større datamengder ble det tydelig at det var mer ressurskrevende å returnere data (*sequence out*) enn å sende parameterdata (*sequence in*).

I del to, maskinvare-sammenlikningene, studerte vi dette funnet nærmere ved å måle *sequence in* og *sequence out* for forskjellige kombinasjoner av iPAQ og arbeidsstasjon. Vi kom frem til at uavhengig av om det er parameter- eller returdata som overføres, er sendingen av disse dataene tyngre enn mottaket. Vi så denne tendensen ved bytte om på iPAQ og arbeidsstasjon, samt gjøre målinger med to iPAQ-er. Dette resultatet sammen med sequence-in/out-funnet fra forrige del, gjorde at vi kunne identifisere den tyngste rollen under overføring av større datamengder. Sending av returdata er den operasjonen som krever desidert mest ressurser, og det å være tjener — den som mottar et metodekall — ved retur av store datamengder, er således den klart tyngste rollen.

Resultater fra lokale målingene med ulik størrelse på overført datamengde viste at iPAQ-en er markant tregere også når vi øker datamengden gradvis fra 0 KB, som tilsvarer enkle metodekall, til 50.000 KB. Bruk av iPAQ gir lavere gjennomstrømning enn arbeidsstasjonen ved lokale målinger, og på-

virker også gjennomstrømningen i en klient-tjener-kombinasjon negativt. Skaleringsevnen til iPAQ ved økende antall objekter på tjenersiden viste seg å være dårligere enn for arbeidsstasjonen. Hvordan iPAQ-en påvirket responstiden som henholdsvis klient og tjener i ulike maskinvarekombinasjoner er omtalt senere.

I den tredje delen, hvor vi gjorde målinger med ulike nettverkstyper, fant vi store forskjeller. Metodekall-tider for Bluetooth ligger høyt over Wi-Fi og lokale målinger. Ved bruk av nettverk generelt, og spesielt med Bluetooth, overskygger responstiden andre forskjeller, som mellom CORBA-implementasjonen og sequence-in/out. For gjennomstrømning var ikke den relative forskjellen i responstid mellom de to trådløse nettverksteknologiene så stor når datamengden økte. Wi-Fis lave gjennomstrømning i forhold til den teoretiske båndbredden, og PPPs komprimering for Bluetooth, gjorde at Wi-Fis responstid prosentvis økte kraftigst, slik at responstidene relativt sett ble mer lik. Den absolutte differanse mellom responstidene ble imidlertid større. Ved dispatcher-målingene så vi at høyere forsinkelse førte til dårligere skalering for responstidene ved økning av antall objekter på tjenersiden.

Vi utførte også en analyse av systemressursforbruket på de involverte maskinene under målingene. CPU-forbruket ved lokale målinger, hvor både klient og tjener kjørte på samme maskin, viste seg å være meget høyt. Her brukte klienten og tjeneren all ledig CPU-tid, og dermed påvirket ytelsen under disse målingene. Ved bruk av nettverk ble CPU-forbruket en god del lavere, særlig med Bluetooth. Her brukes mye tid på I/O-wait grunnet latens i nettverket, og klient og tjener kjører på separate maskiner. Nettverksforbruket var relativt likt for de to CORBA-implementasjonene. Dette rimer godt med at begge bruker IIOP til datainnpakking, og at *Open CORBA Benchmarking* sender samme datamengder uavhengig av hvilken CORBA-implementasjon som testes.

Videre fant vi at MICO genererer et betydelig høyere antall *page faults* enn Orbix/E under *dispatcher*-målingene. Det er derfor nærliggende å anta at MICOs dårlige ytelse har en sterk sammenheng med dette. Det var også en annen forskjell mellom CORBA-implementasjonene under disse målingene. Under initiell testing klarte MICO opptil 37.000 objekter, mens Orbix/E fikk problemer ikke langt over 20.000. Sistnevnte hadde en predefinert øvre grense på 1.000 objekter. Denne versjonen av Orbix er tiltenkt bruk på ressurssvake enheter, noe grensen gir en indikasjon på.

Under analysen kom vi over en del funn vi ikke fikk tid til å se nærmere

på. Blant annet så vi en forskjell i responstider ved *sequence in* og *sequence out*. Vi registrere også at det var tyngre å sende data enn å motta. Disse resultatene har vi ikke funnet en forklaring på. Videre viste *microbenchmarks* at iPAQ-en er nesten like rask som arbeidsstasjonen på trådbytte, på tross av at den er betydelig tregere på andre områder.

Vi har en mistanke om at de dårlige dispatcher-resultatene med iPAQ som klient og arbeidsstasjonen som tjener skyldes en svakhet i *Open CORBA Benchmarking*, hvor en operasjon som etter vår oppfatning ikke burde vært med i tidtakningen kan være årsaken. Det hadde vært interessant å se nærmere på dette, men det krever en større omskriving av de generiske målerutinene i OCB. Et siste funn vi ikke har kunnet forklare er hvorfor skaleringsevnen under *dispatcher*-testen blir merkbart dårligere ved bruk av tregere nettverk.

Kapittel 7

Konklusjon

I denne oppgaven har vi undersøkt bruk av CORBA på håndholdte enheter i trådløse nettverk. Under målingene har vi benyttet måleverktøyet *Open CORBA Benchmarking*, og har ved analysen av måleresultater fokusert på tre områder: *CORBA-implementasjoner*, *maskinvare og maskinvarekombinasjoner* og *nettverk*. For hvert av disse fokusområdene har vi undersøkt variasjoner i ytelse ved å endre konfigurasjonparametre. I sammenlikninger av CORBA-implementasjoner har vi studert *Orbix/E* og *MICO*. Av maskinvare har vi benyttet to iPAQ-er og to ordinære Intel-basert arbeidsstasjon. Operativsystemet *Linux* er benyttet på begge plattformene. Av trådløse nettverksteknologier, har vi benyttet *Wi-Fi* og *Bluetooth*.

Dette kapittelet presenterer resultater fra ytelsesmålingene, og vi trekker noen konklusjoner basert på det vi har kommet frem til under analysen. Videre gir vi en evaluering av hvor langt vi kom i forhold til våre mål, etterfulgt av en oversikt over mulig videre arbeid og områder som kan undersøkes nærmere. Som en avslutning presenterer vi våre egne refleksjoner.

7.1 Resultater

Forsøkene vi har gjennomført viser at det er mulig å benytte CORBA som mellomvare på håndholdte enheter. Vi ser imidlertid at en rekke faktorer gir betydelige utslag på ressurssvake enheter i trådløse nett, og at optimalisering med hensyn på dette kan få stor innvirkning. Blant annet så vi en

klar forskjell mellom ytelsen til Orbix/E og MICO.

Orbix/E er utviklet med tanke på ressurssvake enheter. Dette ga klare utslag under målingene; Orbix/E viste seg å ha gjennomgående lavere responstid ved metodekall, høyere gjennomstrømning og bedre skaleringsevne for antall objekter enn MICO. For sistnevnte ble det generert et uforholdsmessig høyt antall *page faults* under *dispatcher*-testen, noe som kan ha påvirket ytelsen. Våre målinger indikerer at optimalisering med tanke på håndholdte enheter gir en merkbar ytelsesgevinst.

Måleresultater fra de ulike maskinvare-scenariene viste at maskinvarekonfigurasjoner med en kraftig maskin i den ene enden av en forbindelse ga en kraftig ytelsesforbedring i forhold til konfigurasjoner med iPAQ-er i begge ender. Videre fant vi at begge CORBA-implementasjonene brukte lengre tid på å sende data enn å motta dem, og at det er mer krevende å overføre returdata enn parameterdata. Disse funnene lot oss identifisere den tyngste operasjonen i en dataoverføring: sending av returdata. Følgelig får man, under metodekall som returnerer større datamengder, best effekt ytelsesmessig ved å la den kraftigste maskinen fungere som tjener.

iPAQ har vist seg å være en kraftig nok maskinvareplattform til å bruke en mellomvareteknologi som CORBA. Både som klient og tjener klarer iPAQ å behandle større dataoverføringer og et høyt antall objekter. Imidlertid ser vi at den, sett i forhold til arbeidsstasjonen som er brukt under målingene, har dårligere ytelse. Gjennomstrømningen er lavere, og den skalerer dårligere ved økende antall objekter på tjeneren. Dette er i samsvar med spesifikasjoner og målt systemytelse, som viste at arbeidsstasjonen er en kraftigere maskin, og akkurat dette resultatet var derfor ikke overraskende.

Nettverksanalyser viste at både Wi-Fi og Bluetooth legger begrensninger på ytelsen til CORBA-applikasjonene i form av tregere dataoverføring og økt responstid ved metodekall. Særlig Bluetooth innfører høy latens, som ved hyppige metodekall gir seg utslag i redusert ytelse. Relative forskjeller mellom CORBA-implementasjonene blir mindre som følge av latensen i nettverket, og ved bruk av tregere nettverk blir det derfor av mindre betydning om mellomvaren er optimalisert eller ikke. CPU-forbruket påvirkes også av denne latensen, som fører til at mer tid blir brukt på venting i forbindelse med kommunikasjon.

Ved overføring av større datamengder som parameter- eller returverdier, øker responstiden betraktelig ved bruk av nettverk. De målte forskjellene

mellom Wi-Fi og Bluetooth samsvarer med resultater fra gjennomstrømningsmålinger vi har gjort rett mot nettverket. Grunnet komprimering i PPP-laget er gjennomstrømningen ved bruk av Bluetooth sammenliknbar med ytelsen til Wi-Fi. Dette har sammenheng med at dataene som blir overført lar seg komprimere effektivt, og det er derfor rimelig å anta at ulikhetene vil være større ved overføring av komprimerte data, som MPEG-video eller tilsvarende.

7.2 Evaluering av oppgavens mål

Vårt hovedmål har vært å undersøke hvorvidt CORBA kan benyttes på håndholdte enheter. Ved å utføre praktiske eksperimenter, hvor vi har angrepet problemet fra forskjellige vinkler, har vi oppnådd resultater som har gjort oss i stand til å besvare spørsmål vi hadde som mål å få svar på. Vi har definert ulike delmål for å hjelpe oss frem mot hovedmålet, hvorav noen av ulike grunner bare delvis er nådd.

For å ha mulighet til å sammenlikne resultater fra liknende prosjekter med våre egne, har det vært nødvendig å bruke standardiserte metoder for måling av ytelse. Vi har derfor basert oss på retningslinjer fra *Object Management Group*[10], som blant annet spesifiserer at man bør se på responstid, gjennomstrømning og skalering for antall objekter. Måleverktøyet vi benyttet, *Open CORBA Benchmarking*, følger disse retningslinjene, og tilbyr sammenlikning med andres måleresultater gjennom en resultatdatabase.

Bruk av CORBA innfører en viss *overhead*, og vi ønsket derfor å undersøke om ressursknappheten på håndholdte utelukket bruk av slik mellomvare. For å overvåke ressurssituasjonen under ytelsesmålingene har vi skrevet verktøy for å logge CPU-forbruk, bakgrunnslast, nettverksbelastning og minneforbruk. I tillegg har vi gjennomført ytelsesmålinger av maskinvaren. Disse målingene, sammen med loggingen, har gjort oss i stand til å se resultatene i lys av ressurssituasjonen.

For å se nærmere på hvilke begrensninger ressurssituasjonen på håndholdte enheter la på ytelsen, hadde vi som mål å bruke ulik maskinvare under målingene. I tillegg til iPAQ planla vi å bruke Palm V til ytelsesmålinger for å studere CORBA på en enhet med svært begrensede ressurser. At Palm ble forkastet som maskinvareplattform har medført at vi ikke har hatt mulighet til å få det ønskede maskinvareperspektivet på resultatene. iPAQ har vi imidlertid fått benyttet som både CORBA-klient og tjener. Ett

av våre delmål var å undersøke om håndholdte enheter kunne benyttes i begge roller, og vi har gjennom målingene klart å besvare dette.

I utgangspunktet planla vi å gjøre målinger på forskjellige operativsystemer. Ytelsen til programmer som benytter seg av CORBA-funksjonalitet vil påvirkes av OS-et som ligger under, og vi ønsket derfor å sammenlikne målinger gjort på Linux med målinger under Windows CE. Vi fikk ikke sett nærmere på Windows CE, og målet om å sammenlikne ytelsen under forskjellige operativsystemer ble dermed ikke oppfylt.

Vi hadde også som mål å bruke flere ulike typer nettverk under målingene. Trådløse nettverksteknologier vi planla å studere var følgende: *IrDA*, *Wi-Fi*, *Bluetooth* og *GPRS*. For IrDA ville vi prøve å få til målinger både med *Serial IR* og *Fast IR*, og for Wi-Fi ønsket vi å studere både kommunikasjon via en ruter, samt direkte mellom enheter (*Ad-Hoc*-modus). I tillegg var det ønskelig å undersøke Wi-Fi med og uten kryptering. Med Bluetooth ville vi se på forskjellige pakketyper og mot en mobiltelefon som var koblet til nettverket via en GPRS-forbindelse. Som referanse for målinger over de trådløse nettverkstypene planla vi å bruke 100 Mbps-Ethernet.

Vi ble nødt til å prioriterte målinger over to av nettverkstypene: Wi-Fi i *Managed*-modus og Bluetooth. Referansemålinger over 100 Mbps-Ethernet viste seg å være vanskelig grunnet mangel på Linux-kompatible nettverkskort, og vi baserte oss i stedet på lokale målinger uten bruk av nettverk. Med de nettverkstypene vi har benyttet, samt referansemålingene, har vi fått undersøkt bruk av trådløs nettverksteknologi med CORBA som mellomvare. Vi skulle likevel ønske at vi hadde fått anledning til sammenlikne flere nettverkstyper.

Planen var å studere flere forskjellige CORBA-implementasjoner. UIC-CORBA, som var den første implementasjonen vi vurderte, ble forkastet da vi oppdaget at den ikke var kompatibel med CORBA-standarden. Måleverktøyet *Open CORBA Benchmarking* er imidlertid ferdigkonfigurert for flere implementasjoner, blant annet *MICO*, *omniORB*, *Orbix/E*, *VisiBroker* og *TAO*, hvorav vi valgte MICO og Orbix/E til tentative målinger.

Tilpassing av disse implementasjonen for kjøring på iPAQ var mer krevende enn vi hadde forutsett, og vi valgte derfor å begrense oss til ytelsesmålinger på disse to CORBA-implementasjonene. Med én implementasjon tilpasset håndholdte og én beregnet på arbeidsstasjoner, har vi fått et interessant sammenlikningsgrunnlag. Med mer tid til rådighet, hadde det imidlertid vært ønskelig å få inkludert flere CORBA-implementasjoner i

analysen.

7.3 Videre arbeid

Vi begrenset i våre målinger antallet testscenarier. Under analysen har vi imidlertid sett at det hadde vært interessant å gjøre flere målinger på enkelte områder, for således å få mer perspektiv på resultatene. Vi gir her en kort beskrivelse av disse.

- *Andre operativsystemer*
  Under våre målinger benyttet vi kun operativsystemet Linux. Nye målinger med andre OS, som Windows CE og PalmOS, vil kunne gi svar på på hvilke ytelsesmessige begrensninger operativsystemet setter. Grundigere studier av OS-et vil dessuten kunne fastslå hvor begrensningene eventuelt ligger.
- *Annen maskinvare*
  Vi ønsket i utgangspunktet å utføre eksperimenter på to håndholdte; én med relativt lite ressurser og én kraftigere. Den ressurssvake enheten vi planla å benytte, Palm, ble forkastet da vi fant at måleverktøyet OCB ville bli vanskelig, om ikke umulig, å benytte. Orbix/E-dokumentasjonen forteller imidlertid at ORB-en kan brukes som klient på PalmOS[33], og for MICO eksisterer *MICO for the PalmPilot*-prosjektet, som beskriver hvordan MICO kan benyttes med PalmOS. Med et spesialskrevet verktøy mener vi at det vil være mulig å få til en evaluering på denne maskinvareplattformen, noe som kan gi verdifull innsikt i bruk av CORBA ved meget knappe maskinvareressurser.
- *Datakomprimering*
  Ytelsesmålingene våre ble utført med komprimering for Bluetooth, mens ingen komprimering ble brukt ved Wi-Fi-målinger. `ttcp`-målinger viser en betydelig gjennomstrømningsforskjell med og uten komprimering for Bluetooth, og det hadde derfor vært interessant å studere dette nærmere med å foreta ytelsesmålinger av CORBA-implementasjonene gjort med og uten komprimering.

I sammendraget for analysekapittelet kom vi inn på en del funn vi av tidsmessige grunner eller andre årsaker ikke fikk mulighet til å se nærmere på:

- *OCBs måling av tidsforbruk under* dispatcher*-målinger*
  I *Open CORBA Benchmarking*s *dispatcher*-måling blir en operasjon for å finne et tilfeldig objekt på tjeneren inkludert i målingen av tidsforbruk. Dette vil kunne gi seg utslag i lengre responstider, og vi mener at denne operasjonen burde utføres før starten på metodekallet. En forbedring av OCB vil derfor være å utføre omtalte operasjon før måling av tidsforbruk starter. Videre undersøkelser med denne endringen utført kan gi svar på om ytelsesforskjellen mellom klient-tjener-konfigurasjonene iPAQ-arbeidsstasjon og arbeidsstasjon-iPAQ har sammenheng med dette.

- *Sending og mottak av parameter- og returdata*
  Under marshalling-målinger så vi at det var mer ressurskrevende å returnere data enn å sende parameterdata. Videre fant vi at uavhengig av om det er parameter- eller returdata som overføres var sending tyngre enn mottak. Det hadde vært interessant å finne ut mer om hva disse forskjellene skyldes.

- *MICOs page faults*
  Dispatcher-målinger med MICO genererte et høyt antall *page faults*. Dette har sannsynligvis påvirket ytelsen; særlig på en av test-maskinene, hvor vi så en kraftig økning i responstid. Denne tendensen ble ikke observert for Orbix/E, og nærmere sammenlikninger av systemkall i de to CORBA-implementasjonen kan gi nyttig kunnskap i forhold til optimalisering.

- *Trådbytte på iPAQ*
  Microbenchmarks viste at iPAQ-en er meget rask på trådbytte. Sammenliknet med en av de andre test-maskinene er den tilnærmet like rask på denne operasjonen til tross for at den er betydelig tregere på andre områder. Om dette skyldes forskjeller i implementasjonen av operativsystemet, eller om det henger sammen med ulikheter i prosessor-arkitekturen, kan være verdt å studere nærmere.

- *Nettverkets påvirkning av dispatcher-skalering*
  Nettverket hadde kraftig innvirkning på hvordan responstiden skalerte under *dispatcher*-målingene. Uavhengig av antall objekter på tjenersiden gjøres det bare ett metodekall mot et tilfeldig objekt, og det overføres hverken parameter- eller returverdier. For å belyse den observerte tendensen ville det vært nyttig å utføre mer detaljerte målinger av tidsforbruk.

7.4 Refleksjoner

> *...current middleware has been built assuming a set of hardware and software requirements which are suitable for desktop computers, but not for handheld and embedded devices.*
>
> — Manuel Roman

Sitatet ovenfor er hentet fra artikkelen *Reflective Middleware: From Your Desk to Your Hand*[31], og var noe av vår motivasjon for å skrive denne hovedfagsoppgaven. Våre undersøkelser viser at selv om mellomvaren ikke er skrevet med tanke på håndholdte enheter, kan den benyttes, da disse enhetene har blitt kraftigere de siste par årene. Vi har gjort forsøk med en relativt kraftig variant, men både Orbix/E og MICO kan i følge andre brukes som klient under PalmOS på den mer ressurssvake Palm.

Alt i alt må vi si oss fornøyd med det vi har fått til. På tross av at vi ikke har fått den bredden vi først planla, har vi fått forståelse for hvor mye arbeid dette krever. Vi har brukt mye tid på tilpassing og forarbeid til målinger og analyse. Selve målingene, med konfigurasjonsproblemer og mange annullerte målinger, tok også betraktelig lengre tid enn forventet. Vi mener at de resultatene vi har oppnådd er verdifulle, da de har gjort oss i stand til å trekke noen konklusjoner i lys av målene vi satte oss.

Selv om håndholdte enheter nok vil fortsette å bli kraftigere i tiden fremover, mener vi at det vil være et stadig behov for optimalisering og tilpassing. Vi omgir oss med stadig flere elektroniske apparater som kommuniserer med omverdenen, og mellomvare som CORBA tilbyr en standardisert måte for disse å kommunisere på. For at mellomvare skal være attraktivt for produsenter av slike enheter, bør mellomvare imidlertid ikke være for krevende ressursmessig, og optimalisering er derfor viktig.

Tillegg

Tillegg A

Benchmark

Nr	*Scen.*	*ORB*	*Klient*	*Tjener*	*Nettverk*	*Merknad*
1	2	Orbix/E	pc1	pc1	local	
2	8	Orbix/E	ipaq	pc1	Bluetooth	single-slot
3	2	MICO	pc1	pc1	local	
4	1	MICO	ipaq	ipaq	local	
5	8	MICO	ipaq	pc1	Bluetooth	single-slot
6	2	MICO	pc2	pc2	local	
7	2	Orbix/E	pc2	pc2	local	
8	4	MICO	ipaq	pc1	Wi-Fi	
9	4	MICO	ipaq	pc1	Wi-Fi	
10	4	Orbix/E	ipaq	pc1	Wi-Fi	Feil i *seq_out*
11	8	MICO	ipaq	pc1	Bluetooth	single-slot
12	8	Orbix/E	ipaq	pc1	Bluetooth	single-slot
13	3	MICO	ipaq	ipaq2	Wi-Fi	
14	3	Orbix/E	ipaq	ipaq2	Wi-Fi	
15	7	MICO	ipaq	ipaq2	Bluetooth	single-slot
16	7	Orbix/E	ipaq	ipaq2	Bluetooth	single-slot
17	7	MICO	ipaq	ipaq2	Bluetooth	single-slot
18	7	Orbix/E	ipaq	ipaq2	Bluetooth	single-slot
19	10	MICO	ipaq	ipaq2	Bluetooth	5-slot
20	10	Orbix/E	ipaq	ipaq2	Bluetooth	5-slot
21	9	MICO	pc1	ipaq	Bluetooth	single-slot
22	9	Orbix/E	pc1	ipaq	Bluetooth	single-slot
23	5	MICO	pc1	ipaq	Wi-Fi	
24	2	MICO	pc2	pc2	local	
25	5	Orbix/E	pc1	ipaq	Wi-Fi	
26	5	MICO	pc1	ipaq	Wi-Fi	5s logdelay
27	5	MICO	pc1	ipaq	Wi-Fi	Ingen logging
28	2	Orbix/E	pc1	pc1	local	
29	8	Orbix/E	ipaq	pc1	Bluetooth	single-slot
30	1	MICO	ipaq	ipaq	local	Ingen logging
31	1	Orbix/E	ipaq	ipaq	local	Maks ant. obj. 10000

Tabell A.1: Oversikt over alle ytelsestestene vi har gjennomført.

Nr	*Median*	*Q1*	*Q3*	*Snitt*
1	212	211	215	236
2	55.056	54.898	56.217	56.065
3	659	648	663	732
4	4.361	4.325	4.406	6.237
5	61.988	56.179	64.729	63.125
6	323	322	325	352
7	90	90	91	107
8	4.248	4.051	4.429	5.174
9	4.242	4.025	4.421	4.979
10	3.180	2.981	3.358	3.734
11	61.165	56.168	64.199	62.248
12	63.822	57.518	65.080	62.678
13	6.845	6.647	7.081	8.914
14	4.265	4.071	4.474	5.698
15	79.476	73.739	82.458	80.550
16	74.970	73.666	81.288	76.050
17	79.868	73.748	82.523	80.864
18	74.993	73.655	81.314	76.016
19	79.726	73.773	82.556	81.031
20	74.955	73.648	81.293	75.825
21	55.991	54.990	58.060	59.567
22	55.120	54.985	55.995	55.743
23	4.981	4.785	5.165	5.870
24	326	325	328	358
25	3.132	2.932	3.311	3.676
26	4.977	4.780	5.152	5.454
27	4.963	4.773	5.137	5.002
28	210	206	211	237
29	61.808	56.290	64.713	60.912
30	4.356	4.318	4.400	4.384
31	1.206	1.156	1.208	1.627

Tabell A.2: Statistiske data fra invocation-testen. Alleresultatene er angitt i mikrosekunder (μs).

Måling	*Tjener/Klient*	*Antall feil*			
2	Klient	35	av	11.575	(0,30%)
4	Tjener	16	av	5.094	(0,31%)
4	Klient	64	av	5.091	(1,26%)
5	Klient	60	av	6.009	(1,00%)
8	Klient	38	av	4.795	(0,79%)
9	Klient	30	av	4.847	(0,62%)
10	Klient	54	av	6.194	(0,87%)
11	Klient	73	av	6.053	(1,21%)
12	Klient	38	av	12.100	(0,31%)
13	Tjener	22	av	5.556	(0,40%)
13	Klient	42	av	5.553	(0,76%)
14	Klient	90	av	7.108	(1,27%)
15	Tjener	84	av	7.053	(1,19%)
15	Klient	50	av	7.057	(0,71%)
16	Klient	17	av	13.661	(0,12%)
17	Tjener	76	av	7.020	(1,08%)
17	Klient	55	av	7.031	(0,78%)
18	Tjener	1	av	13.574	(0,01%)
18	Klient	11	av	13.581	(0,08%)
19	Tjener	70	av	7.102	(0,99%)
19	Klient	45	av	7.120	(0,63%)
20	Klient	17	av	13.603	(0,12%)
21	Tjener	68	av	6.352	(1,07%)
21	Klinet	1	av	6.741	(0,01%)
22	Klient	1	av	9.089	(0,01%)
23	Tjener	30	av	5.192	(0,58%)
24	Tjener	1	av	7.198	(0,01%)
24	Klient	1	av	7.180	(0,01%)
25	Tjener	1	av	6.624	(0,01%)
25	Klient	10	av	6.934	(0,14%)
26	Tjener	4	av	1.123	(0,36%)
29	Tjener	4	av	12.820	(0,06%)

Tabell A.3: Tilfeller av feil i data for sysusage (32 av 56 målinger).

Tillegg B

Kildekode

B.1 Start av benchmark

B.1.1 runserver.sh

```
1  #!/bin/bash
2
3  # Sync system clock
4  ntpdate -s ulrik
5
6  # Set environment variables
7  . init_env
8
9  # Start netusage in the bg and store PID
10 ./netusage.sh > $LOGDIR/netusage_server.data &
11 NETUPID=$!
12
13 # Start server in the bg
14 ./Server $1 \
15 -ORBInitRef NameService=corbaloc::azrael.uio.no:12345/NameService\
16 2> $LOGDIR/markers_server.data &
17
18 # Start sysusage and wait for it to terminate
19 ./sysusage.sh $! > $LOGDIR/sysusage_server.data
20
21 # Kill netusage
22 kill $NETUPID
```

B.1.2 runclient.sh

```
1  #!/bin/bash
2
3  # Sync system clock
4  ntpdate -s ulrik
5
6  # Set environment variables
7  . init_env
8
9  # Start netusage in the bg and store PID
10 ./netusage.sh > $LOGDIR/netusage_client.data &
11 NETUPID=$!
12
13 # Start client in the bg
14 ./Client $1 \
15 -ORBInitRef NameService=corbaloc::azrael.uio.no:12345/NameService\
16 > $LOGDIR/benchmark.xml 2> $LOGDIR/markers_client.data &
17
18 # Start sysusage and wait for it to terminate
19 ./sysusage.sh $! > $LOGDIR/sysusage_client.data
20
21 # Kill netusage
22 kill $NETUPID
```

B.2 Logging av OS-ressurser

B.2.1 sysusage.sh

```
1  #!/bin/bash
2
3  # Usage: sysusage.sh PID DELAY
4
5  DEFAULT_DELAY=1
6  DELAY=$2
7
8  # Abort if pid isn't specified as first argument
9  if [ ! "$1" ]; then
10     echo "Pid must be specified."
11     exit 1
12 fi
13
14 # Default DELAY = 1 sec
15 if [ ! "$2" ]; then
16     DELAY=$DEFAULT_DELAY
17 fi
18
19 # Abort if process doesn't exist
20 if [ ! -e "/proc/$1" ]; then
21     echo "Process $1 doesn't exist." > /dev/stderr
22     exit 1
23 fi
24
25 # Loop until process dies
26 until [ ! -e "/proc/$1" ]; do
27     echo `date +%s`" "`cat /proc/$1/stat`" "`cat /proc/loadavg`
28     sleep $DELAY
29 done
```

B.2.2 netusage.sh

```
#!/bin/bash

# Usage: netusage.sh DELAY

DEFAULT_DELAY=1
DELAY=$1

# Default DELAY
if [ ! "$2" ]; then
    DELAY=$DEFAULT_DELAY
fi

# Loop forever
while [ 1 ]; do
    echo "Time: "‘date +%s‘
    cat /proc/net/dev
    sleep $DELAY
done
```

B.3 Behandling av måledata

B.3.1 gen_system_plots.pl

```
1  #!/site/perl-5.6.1/bin/perl
2
3  use strict;
4  use Getopt::Long;
5  use Data::Dumper;
6
7  my $file = $ARGV[$#ARGV]; # File name - last cmd line option
8  my $start = 0;
9  my $header = 0;
10 my %options;
11
12 my $pwd = `pwd`;
13 $pwd =~ m&/\w+\.\w+[S|C]-\w+\.\w+[S|C]-(.+)
       -[0-9]+-[0-9]+-[0-9]+-[0-9]+.*&;
14 my $net = $1;
15
16
17 # Command line options
18 &GetOptions('data=s' => \$options{'data'},
19             'time=i' => \$options{'time'}
20             );
21
22 # Abort if the benchmark server and client was run on the same
23 # machine. This only makes sense when the script is called from
24 # another wrapper script.
25 if ($net eq "local" && $options{'data'} eq "n") {
26     exit(0);
27 }
28
29
30 # Die if data file doesn't exist.
31 die("File '$file' doesn't exist.\n") unless -e $file;
32
33
34
35 if ($options{'time'}) {
36     # Use this time as start. (Synchronization with other data file)
37     $start = $options{'time'};
38 }
39
40 if (!$options{'data'}) {
41     # Data type for system, milestones or network should be given from
42     # command line.
```

```
43      die("Type is required. -d {s,m,n}\n");
44  }
45
46  if ($options{'data'} eq "s") {
47      &sysusage();
48  }
49
50  elsif($options{'data'} eq "m") {
51      &milestones();
52  }
53
54  elsif($options{'data'} eq "n") {
55      &net();
56  }
57
58
59  # Function used to print network data
60  sub print_net {
61      my ($bytes, $time, $data_ref) = @_;
62      my %data = %{$data_ref};
63      my $header_printed = 0;
64
65
66      foreach my $key (sort keys(%data)) {
67
68          if (! (($net =~ m/.*bt.*/ && $key eq "ppp0") || ($net eq "wlan" && $key eq "
                eth0") || ($net eq "irda" && $key eq "irda0"))) {
69              next;
70          }
71
72          if (!$bytes->{$key."_trans"}) {
73              $bytes->{$key."_trans"} = $data{$key}{'t_bytes'};
74          }
75
76          if (!$bytes->{$key."_rec"}) {
77              $bytes->{$key."_rec"} = $data{$key}{'r_bytes'};
78          }
79
80          if (!$header) {
81              print "$key(t)\t$key(r)\t";
82          }
83
84
85          # Only print if sent/received values are positive. (Ignore if
86          # counter has been reset.
87          elsif ($data{$key}{'t_bytes'}-$bytes->{$key."_trans"} >= 0 && $data{
                $key}{'r_bytes'}-$bytes->{$key."_rec"}) {
88
89              if (!$header_printed) {
```

```
90                  if ($header) {
91                      print "$time\t";
92                  }
93
94                  else {
95                      print "# Time\t";
96                  }
97                  $header_printed = 1;
98              }
99
100
101
102             print ((($data{$key}{'t_bytes'}-$bytes->{$key."_trans"})*8)/1024);
103             print "\t";
104             print ((($data{$key}{'r_bytes'}-$bytes->{$key."_rec"})*8)/1024);
105             print "\t";
106         }
107
108         $bytes->{$key."_trans"} = $data{$key}{'t_bytes'};
109         $bytes->{$key."_rec"} = $data{$key}{'r_bytes'};
110
111
112     }
113
114     if ($header_printed) {
115         print "\n";
116     }
117 }
118
119
120 # Read network data file and use print_net function to generate
121 # output.
122 sub net {
123     my $time = 0;
124     my @lines;
125     my %bytes;
126
127     open(DATA, "<$file");
128
129     while(<DATA>) {
130
131         # Start of a network log
132         if (m/Time: ([0-9]+)/) {
133
134             # Set start time if it isn't already set.
135             if (!$start) {
136                 $start = $1;
137                 print "# Start = $start\n";
138             }
```

```
139
140             # Calculate offset from the start time.
141             if ($time) {
142                 &print_net(\%bytes, $time-$start, &net_data(\@lines));
143                 $header = 1;
144             }
145
146             $time = $1;
147             @lines = ();
148
149         }
150
151         # The timestamp line is already read. Now read the network
152         # data lines.
153         elsif (m/\s*(\w+:.*)/) {
154             push @lines, $1;
155         }
156
157     }
158
159
160     &print_net(\%bytes, $time-$start, &net_data(\@lines));
161
162     close(DATA);
163
164 }
165
166
167 # Construct a hash from the line with network interface data
168 sub net_data {
169     my ($data_ref) = @_;
170     my %data;
171
172     for my $interf (@{$data_ref}) {
173
174         $interf =~ m/(\w+):\s*([0-9]+)\s+([0-9]+)\s+([0-9]+)\s+([0-9]+)\s
                +([0-9]+)\s+([0-9]+)\s+([0-9]+)\s+([0-9]+)\s+([0-9]+)\s+([0-9]+)\s
                +([0-9]+)\s+([0-9]+)\s+([0-9]+)\s+([0-9]+)\s+([0-9]+).*/;
175
176
177         $data{$1} = {'r_bytes' => $2,
178                      'r_packets' => $3,
179                      'r_errs' => $4,
180                      'r_drop' => $5,
181                      'r_fifo' => $6,
182                      'r_frame' => $7,
183                      'r_compressed' => $8,
184                      'r_multicast' => $9,
185                      't_bytes' => $10,
```

```
186                     't_packets' => $11,
187                     't_errs' => $12,
188                     't_drop' => $13,
189                     't_fifo' => $14,
190                     't_colls' => $15,
191                     't_carrier' => $16,
192                     't_compressed' => $17
193                     };
194     }
195
196     return \%data;
197
198 }
199
200
201 # Create a file with milestone plots
202 sub milestones {
203     open(DATA, "<$file");
204
205     print "# Time\tEvent\t\tName\n";
206     if ($start) {
207         print "# Start: $start\n";
208     }
209
210     while(<DATA>) {
211
212         $_ =~ m/([0-9]+):\s(\w+):\s(\w+)/;
213
214         if (!$start) {
215             $start = $1;
216             print "# Start = $start\n";
217         }
218
219         print $1-$start."\t$2\t$3\n";
220
221     }
222     close(DATA);
223 }
224
225
226 # Create a file with process- and network data
227 sub sysusage {
228     my $jiffies = 0;
229     open(DATA, "<$file");
230
231     print "# Time\tVmSize\tCPU\tLoad\n";
232     if ($start) {
233         print "# Start: $start\n";
234     }
```

```
235
236
237     my $errors = 0;
238     my $lines = 0;
239     while(<DATA>) {
240         my %data;
241
242         $lines++;
243
244         chop($_); # Remove \n at end of the line
245
246         if ($_ eq "") {
247             next;
248         }
249
250         my @elems = split / /, $_;
251
252         # Sometimes the data gets corruptet. Ignore it and
253         # log the event.
254         if ($#elems != 44) {
255             $errors++;
256             next();
257         }
258
259         # The /proc/PID/stat data structure
260         %data = ('ts' => $elems[0],
261                  'pid' => $elems[1],
262                  'comm' => $elems[2],
263                  'state' => $elems[3],
264                  'ppid' => $elems[4],
265                  'pgrp' => $elems[5],
266                  'session' => $elems[6],
267                  'tty_nr' => $elems[7],
268                  'tpgid' => $elems[8],
269                  'flags' => $elems[9],
270                  'minflt' => $elems[10],
271                  'cminflt' => $elems[11],
272                  'majflt' => $elems[12],
273                  'cmajflt ' => $elems[13],
274                  'utime' => $elems[14],
275                  'stime' => $elems[15],
276                  'cutime' => $elems[16],
277                  'cstime' => $elems[17],
278                  'priority' => $elems[18],
279                  'nice' => $elems[19],
280                  # Element 20 is obsolete.
281                  'itrealvalue' => $elems[21],
282                  'starttime' => $elems[22],
283                  'vsize' => $elems[23],
```

```
284                     'rss' => $elems[24],
285                     'rlim' => $elems[25],
286                     'startcode' => $elems[26],
287                     'endcode' => $elems[27],
288                     'startstack' => $elems[28],
289                     'kstkesp' => $elems[29],
290                     'kstkeip' => $elems[30],
291                     'signal' => $elems[31],
292                     'blocked' => $elems[32],
293                     'sigignore' => $elems[33],
294                     'sigcatch' => $elems[34],
295                     'wchan' => $elems[35],
296                     'nswap' => $elems[36],
297                     'cnswap' => $elems[37],
298                     'exit_signal' => $elems[38],
299                     'processor' => $elems[39],
300                     'load1' => $elems[40],
301                     'load5' => $elems[41],
302                     'load15' => $elems[42]
303                     );
304
305         # First timestamp
306         if (!$start) {
307             $start = $data{'ts'};
308             print "# Start = $start\n";
309         }
310
311
312         # Calculate all the jiffies into one total and get the
313         # difference from last sample
314         my $total = ($data{'utime'}+$data{'cutime'}+$data{'stime'}+$data{'
                cstime'});
315         my $sum = $total-$jiffies;
316
317         print $data{'ts'}-$start."\t".($data{'vsize'}/1024)."\t$sum\t$data{'
                load1'}\n";
318
319         $jiffies = $total
320     }
321
322     # Log stats for amount of corrupt data.
323     warn(sprintf "$errors of $lines lines has errors (%3.2f\%).\n",
324          (($errors/$lines)*100));
325
326     close(DATA);
327 }
```

B.3.2 gen_benchmark_plots.pl

```
#!/site/perl-5.6.1/bin/perl

use strict;
use XML::DOM;
use Data::Dumper;


my $file = $ARGV[$#ARGV]; # File name - last cmd line option
my $parser = new XML::DOM::Parser;
my @values = ("min", "avrg", "max");

my $doc = $parser->parsefile ($file);
my $client = $doc->getElementsByTagName("Client")->item(0);
my $server = $doc->getElementsByTagName("Server")->item(0);
my $scale =
    $doc->getElementsByTagName("Timer")->item(0)->
    getAttributeNode("Scale")->getValue;


#################
# Main
#################

&invocation();
&write_data("Sequence In", "sequence_in", "Size");
&write_data("Sequence Out", "sequence_out", "Size");
&write_data("Instances", "instances", "Count");
&write_data("Parallel", "parallel", "Simul");

open(DATA, ">microbenchmarks.txt");
&write_stats("Processor Move", "Size");
&write_stats("Processor Integer");
&write_stats("Processor Float");
&write_stats("Memory Allocation", "Count", (96, 1536, 12288));
&write_stats("Thread Lifecycle", "Count");
&write_stats("Thread Switching");
&write_stats("Thread Locking");
&write_stats("Network Echo", "Size");
close(DATA);


# Dispose DOM tree.
$doc->dispose;


# Return benchmark element
sub get_benchmark {
```

```
48      my ($name, $element) = @_;
49
50      my $benchmarks = $element->getElementsByTagName("Benchmark");
51      my $benchmark;
52
53      # Loop through all Benchmark elements and return if type/name matches.
54      for (my $i = 0; $i < $benchmarks->getLength; $i++) {
55          $benchmark = $benchmarks->item($i);
56          return $benchmark if ($benchmark->getAttributeNode("Type")->getValue
                eq $name);
57      }
58
59  }
60
61
62  # Convert sample to microseconds
63  sub microseconds {
64      my ($tics) = @_;
65
66      return 0 if (!$tics);
67
68      # $tics devided by $scale, mulitiplied by 1.000.000.
69      return sprintf "%.0f", ($tics/$scale)*1000*1000;
70
71  }
72
73
74  #################
75  # Microbenchmarks
76  #################
77
78  sub write_stats {
79      my ($type, $atr, @sizes) = @_;
80      # ^^^^^^^ Used by "Memory allocation" only.
81
82
83
84      # Header
85      print DATA "*****************************
86  * $type\n*****************************\n\n\t\tClient\tServer\n";
87
88      # Counter for first loop.
89      my $j = 0;
90
91      do { # A hack used only by "Memory Allocation"
92
93          my $atr2 = 0;
94
95          if ($#sizes >= 0) {
```

```
96              $atr2 = $sizes[$j];
97          }
98
99          # Client
100         my $data_ref_client = &get_samples($type, $client, $atr, $atr2);
101
102         # Server
103         my $data_ref_server = &get_samples($type, $server, $atr, $atr2);
104
105         # Print size ("Memory Allocation")
106         if (@sizes) {
107             print DATA "Size: $sizes[$j]\n";
108         }
109
110         # For all measurements with the specified attribute.
111         foreach my $key (keys %{$data_ref_server}) {
112             if ($atr) {
113                 print DATA "$atr: $key\n";
114             }
115
116             # For all values (min, average, max)
117             foreach my $value (@values) {
118                 print DATA "\t$value\t$data_ref_client->{$key}->{
                        $value}\t$data_ref_server->{$key}->{$value}\n";
119             }
120
121             print DATA "\n";
122         }
123
124         # End of "Memory Allocation" –loop
125         $j++;
126     } while $j <= $#sizes;
127
128
129 }
130
131
132 # Create data files
133 sub write_data {
134     my ($name, $filename, $atr) = @_;
135
136     my $data_ref = &get_samples($name, $doc, $atr);
137
138     if (!$data_ref) {
139         print "No data for $name\n";
140         return;
141     }
142
143     foreach my $type (@values) {
```

```
144
145         open(DATA, ">$filename\_$type.plot");
146         print DATA "# $name - $type-values\n# $atr\tTime (us)\n";
147
148         # Write data
149         foreach my $key (sort {$a <=> $b} keys(%{$data_ref})) {
150             print DATA "$key\t$data_ref->{$key}->{$type}\n";
151         }
152
153         close(DATA);
154     }
155
156
157 }
158
159
160 # Fetch samples of measurements
161 sub get_samples {
162     my ($benchmark_name, $element, $atr1, $atr2) = @_;
163     my %data;
164
165     my $benchmark = &get_benchmark($benchmark_name, $element);
166
167     if (!$benchmark) {
168         return;
169     }
170
171
172
173     my $measurements = $benchmark->getElementsByTagName("Measurement");
174
175     for (my $i = 0; $i < $measurements->getLength; $i++) {
176         my $measurement = $measurements->item($i);
177
178
179         # Ignore if wrong size-group. Only applies to "Memory Allocation"
180         if ($atr2 && !($measurement->getAttributeNode("Size")->getValue ==
                $atr2)) {
181             next;
182         }
183
184         if ($atr1) {
185             $data{$measurement->getAttributeNode($atr1)->getValue} =
186                 &get_avrg_sample($measurement);
187         }
188
189         else {
190             $data{"unspecified"} =
191                 &get_avrg_sample($measurement);
```

```
192             }
193
194         }
195
196         return \%data;
197     }
198
199
200     # Create a hash of average sample data
201     sub get_avrg_sample {
202         my ($measurement) = @_;
203         my %data;
204         my $samples = $measurement->getElementsByTagName("Sample");
205
206         for (my $i = 0; $i < $samples->getLength; $i++) {
207             my $sample = $samples->item($i);
208             my $values = $sample->getFirstChild->getData;
209
210             $values =~ m/([0-9]+)\s([0-9]+)\s([0-9]+)/;
211
212             # Min values
213             if (!$data{'min'} || $1 < $data{'min'}) {
214                 $data{'min'} = $1;
215             }
216
217             # Max values
218             if (!$data{'max'} || $3 > $data{'max'}) {
219                 $data{'max'} = $3;
220             }
221
222             # Add all values
223             $data{'avrg'} += $2;
224
225             $data{'load_before'} +=
226                 $sample->getAttributeNode("LoadBefore")->getValue;
227             $data{'load_after'} +=
228                 $sample->getAttributeNode("LoadAfter")->getValue;
229         }
230
231
232         $data{'min'} = &microseconds($data{'min'});
233         $data{'avrg'} = &microseconds($data{'avrg'}/$samples->getLength);
234         $data{'max'} = &microseconds($data{'max'});
235
236         $data{'load_before'} = $data{'load_before'}/$samples->getLength;
237         $data{'load_after'} = $data{'load_after'}/$samples->getLength;
238
239         return \%data;
240     }
```

```
241
242 # Round trip
243 sub invocation {
244     my %data; # Hash with measurement data
245     my @data_arr;
246     my $measurements = get_benchmark("Ping Roundtrip", $doc)->
          getElementsByTagName("Measurement");
247     my $res = $measurements->item(0)->getFirstChild->getData;
248
249     # Remove all newlines.
250     $res =~ s/\n/ /g;
251
252     # Samples are separated by whitespace. Split and insert into hash.
253     for my $ticks (split / /, $res) {
254
255         # Ignore if $ticks is empty.
256         if ($ticks eq "") {
257             next;
258         }
259
260         # Insert into data hash.
261         $data{&microseconds($ticks)}++;
262         push @data_arr, &microseconds($ticks);
263     }
264
265     # Only write to files if there is any data.
266     if ($measurements->getLength == 1) {
267         my $filename = "invocation";
268
269
270         # Write data file
271         open (DATA, ">$filename.plot");
272         # Write data
273         foreach my $key (sort {$a <=> $b} keys(%data)) {
274             print DATA "$key\t$data{$key}\n";
275         }
276         close(DATA);
277
278     }
279
280     &invocation_stats(@data_arr);
281
282 }
283
284 # Write invocation stats to file (median, q1, q3, average, min and max)
285 sub invocation_stats {
286     my (@data) = @_;
287     my $sum = 0;
288     my $length = $#data+1;
```

```
289     my ($median, $q1, $q3);
290
291
292     @data = sort {$a <=> $b} @data;
293
294     # Median and quartiles
295     if ($length %2 == 1) {
296         my $med_pos = ($length/2)+0.5;
297         my $q1_pos = $med_pos/2;
298         my $q3_pos = $q1_pos + $med_pos-1;
299
300         #-1 since the first element of the array is numbered "0".
301         $median = $data[$med_pos-1]."\n";
302         $q1 = sprintf "%.0f",($data[$q1_pos-1]+$data[$q1_pos])/2;
303         $q3 = sprintf "%.0f",($data[$q3_pos-1]+$data[$q3_pos])/2;
304     }
305
306     else {
307         my $med_pos = $length/2;
308         my $q1_pos = ($med_pos/2)+0.5;
309         my $q3_pos = $q1_pos + $med_pos;
310
311         #-1 since the first element of the array is numbered "0".
312         $median = sprintf "%.0f",($data[$med_pos-1]+
313                                   $data[$med_pos])/2;
314         $q1 = $data[$q1_pos-1];
315         $q3 = $data[$q3_pos-1];
316     }
317
318     # Sum of all elements (for computation of average)
319     for my $elem (@data) {
320         $sum += $elem;
321     }
322
323     open(FILE, ">benchmark_stats.txt");
324     print FILE "Invocation\n";
325     print FILE "**********\n";
326     print FILE "Median: $median\n";
327     print FILE "Q1: $q1\n";
328     print FILE "Q3: $q3\n";
329     printf FILE "Average: %.0f\n", ($sum/$length);
330     print FILE "Min: $data[0]\n";
331     print FILE "Max: $data[$#data]\n";
332     print FILE "Plot: $q1\t$data[0]\t$data[$#data]\t$q3\t$median\n";
333     close(FILE);
334 }
```

B.3.3 plot.pl

```
1  #!/usr/bin/env perl
2
3  use strict;
4  use Getopt::Long;
5
6  my %options = ('line' => "lines");
7  my @types;
8  my @sysusages;
9  my $plots = "";
10
11 my $ylabel;
12 my $xlabel;
13
14 my %short = ('Stopping' => 'Stp',
15            'Starting' => 'Str',
16            'MeasureProcessor' => 'MP',
17            'MeasureMemory' => 'MM',
18            'MeasureThread' => 'MT',
19            'MeasureSocket' => 'MS'
20            );
21
22 my %xlabels = ('instances' => 'Objects registered by both the server
       and client.',
23             'sequence_in' => 'Length of sequence (bytes)',
24             'sequence_out' => 'Length of sequence (bytes)',
25             'parallel' => 'Numbers of threads executing on the
                  client'
26             );
27
28 &GetOptions("name=s" => \$options{'name'},
29           "type=s" => \@types,
30           "sysusage=s" => \@sysusages,
31           "postscript=s" => \$options{'postscript'},
32           "eps" => \$options{'eps'},
33           "markers" => \$options{'markers'},
34           "line=s" => \$options{'line'},
35           "column=i" => \$options{'column'},
36           "debug" => \$options{'debug'},
37           "xx=i" => \$options{'xx'},
38           "yx=i" => \$options{'yx'},
39           "xm=i" => \$options{'xm'},
40           "ym=i" => \$options{'ym'},
41           "xt=i" => \$options{'xt'},
42           "yt=i" => \$options{'yt'},
43           "help" => \$options{'help'}
44           );
45
```

```
46
47 if ($options{'help'}) {
48     usage();
49     exit(0);
50 }
51
52 sub usage {
53     print << "EOT";
54 Usage: $0 [options]
55 Options:
56     -n, --name [instances, invocation, sequence_in, sequence_out, parallel]
57     -t, --type [min, avrg, max] [mem, cpu, net]
58     -c, --column -t net -c column X
59     -s, --sysusage [client, server]
60     -m, --markers Add markers/milestones
61     -l, --line Plot style [line, points, linespoints]
62     -xx Max range, X
63     -xm Min range, X
64     -yx Max range, Y
65     -ym Min range, Y
66     -xt X-tics
67     -yt Y-tics
68     -p, --postscript [filename]
69     -e, --eps
70     -d, --debug Print plot instructions to stdout.
71     -h, --help This help
72
73 EOT
74 }
75
76
77 # Benchmark plots
78 if ($options{'name'} || $options{'type'}) {
79     die("No name specified.\n") unless $options{'name'};
80
81     if ($options{'name'} eq "invocation") {
82         $plots .= "\"$options{'name'}.plot\" title 'Distribution of
                invocation time'";
83         $ylabel = "Number of occurances";
84         $xlabel = "Response time (us)";
85     }
86
87     else {
88
89         die("No type(s) specified.\n") unless $#types >= 0;
90
91         $ylabel = "Response time (us)";
92         $xlabel = $xlabels{$options{'name'}};
93
```

```
94
95          foreach my $type (@types) {
96              if (!$plots eq "" ) {
97                  $plots .= ", ";
98              }
99              $plots .= "\"$options{'name'}_$type.plot\" title '$options
                    {'name'} $type'";
100         }
101     }
102 }
103
104 # System usage
105 else {
106     foreach my $sysusage (@sysusages) {
107         if (!$plots eq "") {
108             $plots .= ", ";
109         }
110
111         die("No type(s) specified.\n") unless $#types >= 0;
112
113         my $columns;
114         my $name;
115         my $file;
116
117         $xlabel = "Benchmark running time (s)";
118
119         if ($types[0] eq "mem") {
120             $file = "sysusage";
121             $columns = "1:2";
122             $name = "Memory usage";
123             $ylabel = "Memory usage (kB)";
124         }
125
126         elsif ($types[0] eq "cpu") {
127             $file = "sysusage";
128             $columns = "1:3";
129             $name = "CPU usage";
130             $ylabel = "CPU usage (%)";
131         }
132
133         elsif ($types[0] eq "load") {
134             $file = "sysusage";
135             $columns = "1:4";
136             $name = "Load";
137             $ylabel = "Load";
138         }
139
140         elsif ($types[0] eq "net") {
141
```

```
142             if (!$options{'column'}) {
143                 usage();
144                 exit(1);
145             }
146
147             $file = "netusage";
148             $columns = "1:".($options{'column'}+1);
149             $ylabel = "Net usage (kbit/s)";
150             my @types;
151
152             open(NET, "head -2 $file\_$sysusage.plot|");
153             while(<NET>) {
154
155                 if ($_ =~ m/.*Time.*/) {
156                     my $val = $_;
157                     $val =~ s/.*Time\W//;
158                     chop($val);
159                     @types = split /\t/, $val;
160                 }
161             }
162
163
164             close(NET);
165
166             $name = "Net usage";
167
168             if ($#types >= $options{'column'}-1) {
169                 $name .= "  ".$types[$options{'column'}-1];
170
171             }
172
173         }
174
175         else {
176             die("No such type (cpu or mem).\n");
177         }
178
179         # Markers
180         if ($options{'markers'}) {
181             $plots .= "'$file\_$sysusage.plot' using $columns title '
                        $sysusage: $name', 'markers_$sysusage.plot' using
                         1:(1) axes x1y2 with impulse";
182         }
183
184         else {
185             $plots .= "\"$file\_$sysusage.plot\" using $columns title '
                        $sysusage: $name'";
186         }
187     }
```

```
188 }
189
190
191 # Open pipe to process and write
192 if (!$options{'debug'}) {
193     open(PLOT, "| gnuplot -persist") or die("Could't start gnuplot.\n"
            );
194 }
195
196 else {
197     open(PLOT, ">&STDOUT");
198 }
199
200 print PLOT "set data style $options{'line'}\n";
201
202 # Postscript output
203 if ($options{'postscript'}) {
204
205     if ($options{'eps'}) {
206         print PLOT "set terminal postscript eps\n";
207     }
208
209     else {
210         print PLOT "set terminal postscript\n";
211     }
212
213     print PLOT "set output \"$options{'postscript'}\"\n";
214 }
215
216 # Range
217 if ($options{'xx'} || $options{'xm'}) {
218     print PLOT "set xrange [$options{'xm'}:$options{'xx'}]\n";
219 }
220
221 if ($options{'yx'} || $options{'ym'}) {
222     print PLOT "set yrange [$options{'ym'}:$options{'yx'}]\n";
223 }
224
225 # Tics
226 if ($options{'xt'}) {
227     print PLOT "set xtics $options{'xt'}\n";
228 }
229
230 if ($options{'yt'}) {
231     print PLOT "set ytics $options{'yt'}\n";
232 }
233
234
235
```

```
236
237 # Labels for axis
238 print PLOT "set ylabel '$ylabel'\n";
239 print PLOT "set xlabel '$xlabel'\n";
240
241
242
243 # Title
244 my $pwd = `pwd`;
245 $pwd =~ m&/(\w+)\.(\w+)(S|C)-(\w+)\.(\w+)(S|C)-(.+)-([0-9]+)-([0-9]+)
        -([0-9]+)-([0-9]+).*&;
246
247 my %md = ('net' => $7,
248                 'date' => "$10.$9.$8",
249                 'no' => $11
250                 );
251
252 if ($3 eq "S") {
253     $md{'server-orb'} = $1;
254     $md{'server'} = $2;
255     $md{'client-orb'} = $4;
256     $md{'client'} = $5;
257 }
258
259 else {
260     $md{'server-orb'} = $4;
261     $md{'server'} = $5;
262     $md{'client-orb'} = $1;
263     $md{'client'} = $2;
264 }
265
266
267
268 print PLOT "set title \"Client: $md{'client-orb'}\@$md{'client'}
        Server: $md{'server-orb'}\@$md{'server'} Net: $md{'net'} Date
        : $md{'date'} No: $md{'no'}\"\n";
269
270 if (!$plots) {
271     usage();
272     exit(1);
273 }
274
275
276 # The Plot
277 print PLOT "plot $plots\n"; # Plot
278
279
280
281 close(PLOT);
```

B.3.4 gen_postscript.sh

```
#!/bin/bash

PS_DIR=$PWD/"ps"
PLOT_DIR="plots"

if [ ! -e $PS_DIR ]; then
    mkdir $PS_DIR
fi

cd $PLOT_DIR

plot.pl -n instances -t min -t avrg -t max -l linespoints -p $PS_DIR/instances_mam.ps
plot.pl -n instances -t min -t avrg -l linespoints -p $PS_DIR/instances_ma.ps

plot.pl -n invocation -p $PS_DIR/invocation.ps -l points

if [ -e "parallel_min.plot" ] ; then
    plot.pl -n parallel -t min -t avrg -t max -l linespoints -p $PS_DIR/parallel_mam.ps
    plot.pl -n parallel -t min -t avrg -l linespoints -p $PS_DIR/parallel_ma.ps
fi

plot.pl -n sequence_in -t min -t avrg -t max -l linespoints -p $PS_DIR/sequence_in_mam.ps
plot.pl -n sequence_in -t min -t avrg -l linespoints -p $PS_DIR/sequence_in_ma.ps

plot.pl -n sequence_out -t min -t avrg -t max -l linespoints -p $PS_DIR/sequence_out_mam.ps
plot.pl -n sequence_out -t min -t avrg -l linespoints -p $PS_DIR/sequence_out_ma.ps

plot.pl -s client -t mem -m -p $PS_DIR/sysusage_client_mem.ps
plot.pl -s client -t cpu -m -p $PS_DIR/sysusage_client_cpu.ps -ym 0 -yx 120

plot.pl -s server -t mem -m -p $PS_DIR/sysusage_server_mem.ps
plot.pl -s server -t cpu -m -p $PS_DIR/sysusage_server_cpu.ps -ym 0 -yx 120


plot.pl -s client -t net -c 1 -p $PS_DIR/netusage_client_t.ps
plot.pl -s client -t net -c 2 -p $PS_DIR/netusage_client_r.ps

plot.pl -s server -t net -c 1 -p $PS_DIR/netusage_server_t.ps
plot.pl -s server -t net -c 2 -p $PS_DIR/netusage_server_r.ps

plot.pl -s client -t load -yx 3 -p $PS_DIR/load_client.ps
plot.pl -s server -t load -yx 3 -p $PS_DIR/load_server.ps
```

```
43
44
45 ~larsar/bin/i2ps microbenchmarks.txt > $PS_DIR/microbenchmarks.ps
46
47 cd ..
```

B.3.5 gen_plots.pl

```
1  #!/site/perl-5.6.1/bin/perl
2
3  use strict;
4
5  my $plot_dir = "plots";
6  my $ROOT = "/hom/larsar/hfag/src/scripts";
7
8
9  # Plots directory
10 if (! -e "$plot_dir") {
11     mkdir("$plot_dir");
12 }
13
14 # benchmark.xml
15 if (-e "benchmark.xml") {
16     print "Benchmark\n";
17     `cd $plot_dir; $ROOT/gen_benchmark_plots.pl ../benchmark.xml`;
18 }
19
20 my $time;
21
22 for my $file ("sysusage_server.data", "sysusage_client.data") {
23
24     if (-e $file) {
25         $time = `head -2 $file`;
26         $time =~ m/([0-9]+)\s+.*/;
27         $time = $1;
28         last;
29     }
30 }
31
32
33 print "Sysusage, server\n";
34 `$ROOT/gen_system_plots.pl -d s sysusage_server.data > $plot_dir/sysusage_server.
       plot`;
35 print "Sysusage, client\n";
36 `$ROOT/gen_system_plots.pl -t $time -d s sysusage_client.data > $plot_dir/
       sysusage_client.plot`;
37 print "Netusage, server\n";
```

```
38 '$ROOT/gen_system_plots.pl -t $time -d n netusage_server.data > $plot_dir/
       netusage_server.plot';
39 print "Netusage, client\n";
40 '$ROOT/gen_system_plots.pl -t $time -d n netusage_client.data > $plot_dir/
       netusage_client.plot';
41 print "Markers, server\n";
42 '$ROOT/gen_system_plots.pl -t $time -d m markers_server.data > $plot_dir/
       markers_server.plot';
43 print "Markers, client\n";
44 '$ROOT/gen_system_plots.pl -t $time -d m markers_client.data > $plot_dir/
       markers_client.plot';
45 'chmod -R g+rXw $plot_dir'
```

Tillegg C

Dataformat, OCB-resultatfil

```
1  ------------
2  Documentation for the benchmark.
3  1.01
4  ------------
5
6  <Results
7      VersionMinor="1"
8  >
9    <Client>
10     <Identity>
11       <Client
12           Threaded=Boolean Says "Yes" when the client can use
13                                 multiple threads, "No" otherwise.
14
15                                 When the client does not use multiple
16                                 threads, all benchmarks that require
17                                 invocations from multiple threads are
18                                 skipped.
19       >
20       </Client>
21     </Identity>
22   </Client>
23   <Server>
24     <Identity>
25       <Server
26           Threaded=Boolean Says "Yes" when the server can use
27                                 multiple threads, "No" otherwise.
28       >
29       </Server>
30     </Identity>
31   </Server>
32 </Results>
```

```
33
34 ----------------
35  1.00 01/04/11
36 ----------------
37
38 <Results
39     VersionMajor="1" Incremented on incompatible change.
40     VersionMinor="0" Incremented on compatible change.
41 >
42   <Configuration
43       Uniquifier=String Random string uniquifying results.
44       Local=Boolean Says "Yes" when server and client are
45                                   on the same host, "No" otherwise.
46   />
47
48   <Client
49       VersionMajor=Number Major benchmark version.
50       VersionMinor=Number Minor benchmark version.
51   >
52       Text Client identification.
53
54     <Identity>
55       <Processor
56           Vendor=String Processor vendor identification, so
57                                   far the strings that can appear here
58                                   besides "Unknown" are "AMD", "Centaur",
59                                   "Cyrix", "Intel", "NexGen", "Rise",
60                                   "TransMeta", "UMC".
61           Family=String Processor family identification, so
62                                   far the strings that can appear here
63                                   besides "Unknown" are "80x86", "Sparc".
64           Model=String Processor model identification. This
65                                   is a relatively short identification
66                                   of the processor, but there is no
67                                   standard format to it.
68           Clock=Integer The approximate processor clock in MHz.
69           Number=Integer The number of active processors.
70       >
71           Text Processor identification.
72       </Processor>
73       <Memory
74           PhysicalFree=Integer Free physical memory in bytes.
75           PhysicalTotal=Integer Total physical memory in bytes.
76           VirtualFree=Integer Free virtual memory in bytes.
77           VirtualTotal=Integer Total virtual memory in bytes.
78       />
79       <Timer
80           Scale=Integer How many timer units are in one second.
81           Granularity=Integer What is the smallest timer increment.
```

```
/>
<System
    Vendor=String Operating system vendor identification,
                        so far the strings that can appear here
                        besides "Unknown" are "Microsoft",
                        "Opensource", "Sun".
    Family=String Operating system family identification,
                        so far the strings that can appear here
                        besides "Unknown" are "Unix", "Windows".
    Model=String Operating system model identificatin.
                        This is a relatively short identification
                        of the operating system, but there is no
                        standard format to it.
    VersionMajor=Integer The operating system major version number.
    VersionMinor=Integer The operating system minor version number.
>
    Text Operating system identification.
</System>
<Compiler
    Vendor=String Compiler vendor identification, so far the
                        strings that can appear here besides "Unknown"
                        are "Microsoft", "Opensource".
    Family=String Compiler family identification, so far the
                        strings that can appear here besides "Unknown"
                        are "C++".
    Model=String Compiler model identificatin. This is a
                        relatively short identification of the
                        compiler, but there is no standard format to
                        it.
    Optimized=Boolean Says "Yes" when some sort of optimization
                        has been turned on, "No" otherwise.
    VersionMajor=Integer The compiler major version number.
    VersionMinor=Integer The compiler minor version number.
>
    Text Compiler identification.
</Compiler>
<Broker
    Vendor=String Broker vendor identification.
    Family=String Broker family identification.
    Model=String Broker model identificatin.
    VersionMajor=Integer The compiler major version number.
    VersionMinor=Integer The compiler minor version number.
>
    Text Broker identification.
</Broker>
<Protocol
    Vendor=String Protocol vendor identification.
    Family=String Protocol family identification.
    Model=String Protocol model identificatin.
```

```
131         VersionMajor=Integer The protocol major version number.
132         VersionMinor=Integer The protocol minor version number.
133       >
134         Text Protocol identification.
135       </Protocol>
136       <Client
137         Connection=String Connection mode identification, the strings
138                                   that can appear here besides "Unknown" are:
139                                     "Per Server" when the client opens one
140                                       connection per server,
141                                     "Per Thread" when the client opens one
142                                       connection per thread.
143         Invocation=String Invocation mode identification, the strings
144                                   that can appear here besides "Unknown" are:
145                                     "Exclusive" when each invocation owns the
146                                       connection exclusively,
147                                      "Shared Passive" when invocations can share
148                                        connections but only invoking threads are
149                                        used to service it,
150                                     "Shared Active" when invocations can share
151                                       connections and extra threads are used to
152                                       service it.
153       >
154         Text Client invocation mode identification.
155       </Client>
156     </Identity>
157     <Benchmark
158        Type=String Type of the benchmark.
159     >
160       <Measurement
161         Size=Integer Size of the benchmark op, if any.
162         Count=Integer Count of the benchmark op, if any.
163         Simul=Integer Simultaneousness of the benchmark op, if any.
164       >
165         <Sample
166           LoadBefore=Integer Load of the host just before measurement.
167           LoadAfter=Integer Load of the host just after measurement.
168         >
169           Integer Minimum value observed.
170           Integer Average value observed.
171           Integer Maximum value observed.
172         </Sample>
173       </Measurement>
174     </Benchmark>
175   </Client
176   <Server>
177     <Identity>
178       <Server
179         Connection=String Connection mode identification, the strings
```

```
                                    that can appear here besides "Unknown" are:
                                      "Single" when the server can handle just one
                                        connection,
                                      "Multiple" when the server can handle
                                        multiple connections.
          Invocation=String Invocation mode identification, the strings
                                    that can appear here besides "Unknown" are:
                                      "Exclusive" when the server can process just
                                        one invocation at any time,
                                      "Per Client" when the server can process one
                                        invocation for each connected client at
                                        any time,
                                      "Per Object" when the server can process one
                                        invocation for each servant at any time,
                                      "Per Request" when the server can process an
                                        arbitrary number of invocations at any
                                        time,
                                      "Thread Pool" when the server can process a
                                        certain limited number of invocations at
                                        any time.
        >
          Text Server invocation mode identification.
        </Server>
      </Identity>
    </Server>
    <Session>
    </Session>
</Results>

------------
Copyright (C) Petr Tuma <petr.tuma@mff.cuni.cz>
Adam Buble <adam.buble@mff.cuni
------------
```

Tillegg D

Nettreferanser

D.1 Maskinvare

- CNET Maskinvareguide
  `http://computers.cnet.com/`
- Ethernet-historie
  `http://www.techfest.com/networking/lan/ethernet1.htm`
- Hastighet på seriell kommunikasjon med iPAQs Bluetooth CSR-brikke
  `http://www.handhelds.org/pipermail/ipaq/2002-August/015227.html`
- Intel
  `http://www.intel.com/`
- Intels XScale-arkitektur
  `http://www.intel.com/design/intelxscale/index.htm`
- Pressemelding ang. lansering av SA-110
  `http://www.mitronet.com/chipdir/oth/strongarm.txt`
- Palm-historie
  `http://www.palm.com/about/corporate/timeline.html`
- Supportside for Palm, U.S. Robotics og Planet Project
  `http://www.3com.no/otherurls/`
  Oppdelingen av 3Com til mindre firmaer .

- USB-historie
  `http://www.computerworld.com/hardwaretopics/hardware/story/0,10801,71063,00.html`

D.2 Programvare

- ELinOS
  `http://www.elinos.com/`
- Ethereal
  `http://www.ethereal.com/`
- Familiar Linux
  `http://www.handhelds.org/familiar/`
- Gnuplot
  `http://www.gnuplot.info/`
- LEM
  `http://www.linux-embedded.com/`
- Lineos Embedix Plus PDA
  `http://www.lineo.com/products/embedix_plus/shd/`
- Linu@
  `http://www.mizi.com/en/prod/embed/linuette-mobile.htm`
- LinuxDA
  `http://www.linuxda.com/`
- MICO
  `http://www.mico.org/`
- MICO/MT
  `http://micomt.sourceforge.net/`
- MICO for the PalmPilot
  `http://www.mico.org/pilot/`
- MontaVista Linux
  `http://www.mvista.com/products/`

- Netspec
  http://www.ittc.ukans.edu/netspec/
- Netspec CORBA-modul
  http://www.ittc.ukans.edu/projects/corba/pmo/
- Open CORBA Benchmarking
  http://nenya.ms.mff.cuni.cz/~bench/
- Orbix/E
  http://www.iona.com/products/orbix-e.htm
- PalmOS 5
  http://www.palmsource.com/palmos/intro_os5.html
- Orbit
  http://orbit-resource.sourceforge.net/
- POSE
  http://www.palmos.com/dev/tools/emulator/
- Pressemelding om samarbeid mellom Microsoft og Intel (Xscale og Media support
  http://www.intel.com/pressroom/archive/releases/20010522net_a.htm
- Sockets i PalmOS
  http://www.palmos.com/dev/support/docs/webclipping/WCAErrorApdx.html
- μClinux
  http://www.uclinux.org/

D.3 Prosjekter

- Douglas Schmidts Research on High-performance CORBA
  http://siesta.cs.wustl.edu/~schmidt/corba-research-performance.html
- MULTE
  http://www.unik.no/~multe/

- UMNS
  http://www.uninett.no/testnett/umns/

D.4 Standarder

- CORBA History
  http://www.omg.org/gettingstarted/history_of_corba.htm
- ISO 14750 - OMG IDL
  http://www.iso.ch/cate/d25486.html
- OMG IDL
  http://www.omg.org/gettingstarted/omg_idl.htm
- OMG Specifications and Process
  http://www.omg.org/gettingstarted/overview.htm#OMGspecs
- WECA Faq
  http://www.wi-fi.com/pdf/20011018_FAQ.pdf

D.5 Diverse

- Dokpros norskordbok
  http://www.dokpro.uio.no/ordboksoek.html

Bibliografi

[1] Object Management Group. *The Common Object Request Broker: Architecture and Specification 3.0*, July 2002.

[2] Neil Rhodes and Julie McKeehan. *Palm Programming: The Developer's Guide*. O'Reilly & Associates, Sebastopol, California, 1998.

[3] Andrew T. Campbell, Geoff Coulson, and Michael E. Kounavis. Managing complexity: Middleware explained. *IT Professional, IEEE Computer Society*, 1999.

[4] David E. Bakken. Middleware. *Encyclopedia of Distributed Computing*, 2001.

[5] Andreas Vogel, Bhaskar Vasudevan, Maira Bejamin, and Ted Villalba. *C++ Programming with CORBA*. John Wiley & Sons, Inc, 1999.

[6] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[7] Object Management Group. *ISO 14750 - Interface Definition Language*, 1999.

[8] Irfan Pyarali and Douglas C. Shmidt. An overview of the corba portable object adapter. *ACM StandardView*, 1999.

[9] Anirudda S. Gokhale and Douglas C. Schmidt. Measuring and optimizing corba latency and scalability over high-speed networks. *IEEE Transaction on Computers*, 47(4), April 1998.

[10] Benchmark PSIG. *White Paper on Benchmarking v1.0*. Object Management Group, 1999.

[11] Petr Tuma and Adam Buble. On benchmarking object-oriented communication middleware. 2001.

[12] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[13] Matthew S. Gast. *802.11 Wireless Networks, The Definitive Guide*. O'Reilly & Associates, 2002.

[14] Jennifer Bray and Charles F. Sturman. *Bluetooth: connect without cables*. Prentice-Hall, 2001.

[15] Bluetooth SIG. *Bluetooth Specification v1.1*, 2001.

[16] Infrared Data Association. *IrDA Serial Infrared Physical Layer Specification v1.4*, 2001.

[17] Tanenbaum Andrew S. *Computer Networks*. Prentice Hall, 1996.

[18] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.

[19] Anirudda S. Gokhale and Douglas C. Schmidt. Evaluating corba latency and scalability over high-speed atm networks. Washington University, Department of Computer Science, May 1997. ICDCS.

[20] Douglas C. Schmidt. Evaluating architectures for multi-threaded corba object request brokers.

[21] Aniruddha Gokhale and Douglas C. Schmidt. Optimizing a corba iiop protocol engine for minimal footprint multimedia systems, 1999.

[22] Anil Gopinath, Sridhar Nimmagadda, Chanaka Liyanaarachchi, Douglas Niehaus, and A. Kaushal. Performance measurement of corba endsystems. *Usenix*, 1999.

[23] Sridhar Nimmagadda, Chanaka Liyanaarachchi, and Douglas Niehaus. Performance patterns: Automated scenario-based orb performance evaluation. *Usenix*, 1999.

[24] Làszlò Böszörményi, Andreas Wickner, and Harald Wolf. Performance evaluation of object oriented middleware. 1999.

[25] Làszlò Böszörményi, Andreas Wickner, and Harald Wolf. Performance evaluation of object oriented middleware - development of a benchmark toolkit. 1999.

[26] *Microsoft Windows CE Programmer's Guide*. Microsoft Press, 1998.

[27] Chris Muench. *The Windows CE Technology Tutorial*. Addison Wesley, 2000.

[28] Roelof J.T. Jonkman. Netspec: Philosophy, design and implementation. Master's thesis, B.S.Co.E. Hogeschool Enschede, Nederland, 1994.

[29] Anil Gopinath. Performance measurement and analysis of real-time corba endsystems. Master's thesis, University of Kansas, 1993.

[30] Petr Tuma and Adam Buble. Technical report on open corba benchmarking. *Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.

[31] Manuel Román, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online Journal*, 2001. Special Issue on Reflective Middleware.

[32] Manuel Román. *UIC - CORBA Personality Users Guide, Client Side Description*.

[33] IONA Technologies. *Orbix/E for C, C++ and Java*, 3 2002.

[34] Raj Jain. *The Art of Computer Systems Performance Analysis*, chapter 3. John Wiley & Sons, Inc, 1991.

[35] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[36] Gary. Wright and W. Richad Stevens. *TCP/IP Illustrated, Volume 2*, chapter 10. Addison-Wesley Publishing Company, 1995.

[37] W. Richad Stevens. *TCP/IP Illustrated, Volume 2*, chapter 18. Addison-Wesley Publishing Company, 1994.

[38] Philip Ezolt. A study in malloc: A case of excessive minor faults. In *Proceedings of the 5th Annual Linux Showcase & Conference*, pages 153–164. USENIX Association, November 2001.